ACS

Color 320 x 240 LCD

Basic Programming

User's Manual

8 September 2014

ACS
Ackerman Computer Sciences

On The Cutting Edge of Technological Evolution

6233 E. Sawgrass Rd • Sarasota, FL. 34240• (941)377-5775 FAX(941)378-4226
www.acscontrol.com

Table of Contents

Welcome Newbies!

If you don't know anything about programming computers, relax – we will try and make it simple. Using this manual as a guide, you'll be able to interact and use your ACS Color LCD Display right away. So sit down and spend a couple of hours with this manual and the display. Interact with it. Get comfortable with them. Once you learn how to program, the sky is the limit.

… And Hello Programmers

If you already know how to program, then turn to the ACS Basic Reference and ACS Basic Examples sections toward the end of this manual. There are summaries of the ACS Basic commands, statements, functions, operators and other programming elements that are supported.

To Get Started…

In order to start using the Color LCD you will need a power source and optionally a communications device. The power source provides the power that the display requires for its operation. The communications device allows you to interact with the built-in ACS Basic language to develop and test your applications.

Powering the Color LCD

The Color LCD can be powered in three different ways:

The display requires a source of power to operate. There are three ways to power the display:

1. Using a couple of pins on the SERIAL connector to connect an external power supply.
2. Using a USB connection to supply power.
3. Using the optional Power Over Ethernet (POE) module and an external Ethernet power injection module.

WARNING: Do not install the USB_POWER jumper if powering the display through the SERIAL connector or external Ethernet power injection module. High voltage will be injected back through your USB cable to the host computer.

Let's look at each of these options in more detail.

Via SERIAL Connector with ACS Power Injector

This consists of a back-to-back pair of DB-9 serial connectors with a wall transformer. The Color LCD can be powered through two pins on its SERIAL connector. This injector supplies that power while allowing the other serial connector pins to pass through to your application cabling and hardware.

ACS LCD Power Injector

Via USB DEVICE Connector with USB Cable

This consists of a USB A to Micro B cable that connects the Color LCD’s DEVICE connector to your PC. The USB_POWER jumper by the USB DEVICE connector must be installed.

WARNING: Do not install the USB_POWER jumper if powering the display through the SERIAL connector or external Ethernet power injection module. High voltage will be injected back through your USB cable to the host computer.

USB A to Micro B Cable

Via ETHERNET Connector with Power Over Ethernet

This requires the optional POE module be installed on the Color LCD and an IEEE802.3af Ethernet power injector cabled to the ETHERNET jack on the display.

PWR128RA 19W Power Over Ethernet Injector

The connection supports Power Over Ethernet (POE) if the optional POE module is installed on the controller and can be used to power the display with remote power injection. The POE support is IEEE802.af compliant and provides a Class 0 signature. Both DC power on Spares (mode B) and DC power on Data (mode A) operation is supported:

ETHERNET Pin #	POE DC Power on Spares		POE DC on Data	
	MDI Signal	MDIX Signal	MDI Signal	MDIX Signal
1	TX+	RX+	TX+ PSE+	RX+ PSE+
2	TX-	RX-	TX- PSE+	RX- PSE+
3	RX+	TX+	RX+ PSE-	TX+ PSE-
4	PSE+	PSE+		
5	PSE+	PSE+		
6	RX-	TX-	RX- PSE-	TX- PSE-
7	PSE-	PSE-		
8	PSE-	PSE-		

More information about powering the display can be found in the Color LCD 320x240 Display Terminal User’s Manual.

Communicating with the Color LCD

There are several options for communicating with the Color LCD. These are listed in order of decreasing preference with the last two somewhat limiting your ability to program the display.

Serial Connection via RS-232

This requires either a stand-alone ANSI terminal device, or a PC running an ANSI terminal emulator program. The connection is made between the Color LCD and the terminal device using a cable between the SERIAL connector and the communications port on the terminal or PC. The PC communication port can be built-in or provided with an external USB Serial Adaptor. ***(See the Color 320x240 LCD Display Terminal User's Guide appendix Wiring Harness Diagram)***

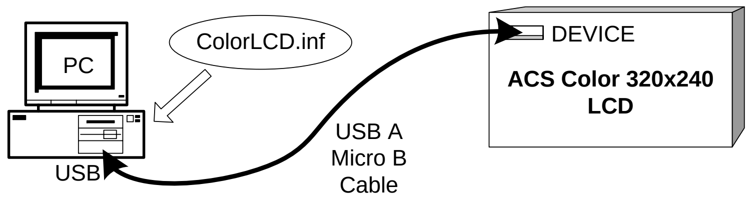
Serial Connection via USB

This requires a PC running an ANSI terminal emulator program. The LCD display can be connected as a USB serial communications device. A Micro B USB DEVICE connector is provided and the display can be connected to a PC with a USB A to Micro B cable.

A ColorLCD.inf file is available that identifies the Color LCD as a virtual serial port device that implements the Communications Device Class. (A Linux USB CDC configuration file is also available)

WARNING: Do not install the USB_POWER jumper if powering the display through the SERIAL connector or external Ethernet power injection module. High voltage will be injected back through your USB cable to the host computer.

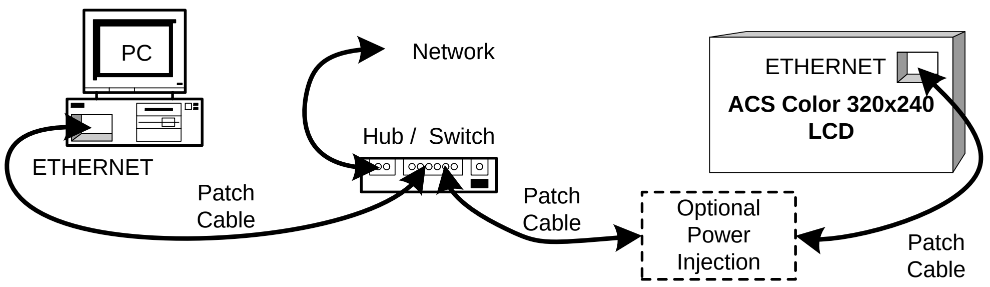
Connect the **USB A to Micro B cable** to the Color LCD **DEVICE** connector. Connect the other end of the cable into a USB port on your PC.



Windows will indicate that it has found new hardware and will eventually prompt for the location of a driver for the Color LCD. Browse to the location of the ColorLCD.inf file that you have downloaded and select it. Windows should now finish installing the new hardware and it should be ready to use. The COM port identifier that Windows will assign to the Color LCD will depend upon what other communications devices are present in the system and can be determined using the Device Manager.

TCP/IP Raw Connection via Ethernet

This requires a PC running an ANSI terminal emulator program that is capable of communicating using TCP/IP Raw sockets – Hyperterminal or the PuTTY program are examples. The LCD display can be connected as an Ethernet device. A standard RJ-45 connector is provided and it can be connected to a network with a standard Ethernet cable – either straight or crossover, detection and correction is automatic via HP Auto MDI/MDI-X configuration. The network speed can be either 10 or 100 mbps with auto link negotiation. A link activity indicator is provided on the ETHERNET jack.



The LCD display supports a configurable MAC address and configurable static IP address and IP mask. Communication is performed using TCP/IP Raw Sockets with a configurable port address.

Terminal Emulator Programs

Your PC may have a terminal emulator program available. Windows XP systems have a program called Hyperterminal installed. It should be located at:

Start, All Programs, Accessories, Communications, Hyperterminal.

A slightly better version of this program, Hyperterminal PE (Private Edition), is also available for purchase from Hilgraeve:

Hyperterm PE: http://www.hilgraeve.com/hyperterminal/

The Hyperterminal PE edition can communicate with the Color LCD using your PC's serial or USB port or with TCP/IP Raw Sockets.

The following terminal emulator programs are also available for free download:

Tera Term: http://ttssh2.sourceforge.jp

PuTTY: http://www.chiark.greenend.org.uk/~sgtatham/putty/

PS/2 Keyboard

This requires the optional ACS-COLOR-LCD-PS2-IO adaptor installed onto the Color LCD EXP connector, and that the display is powered from either the serial or Ethernet power injector. This option uses the display directly as the interactive device with an external PS/2 keyboard. This configuration would be hard to use due to the limited number of characters that can be displayed or if you're developing a graphics program.

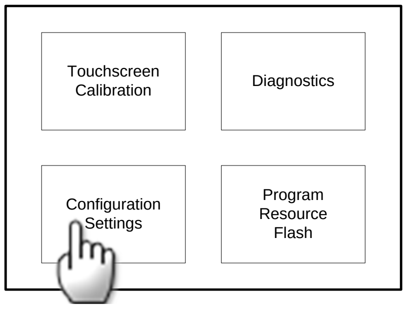
Pop-up Keypad

This option doesn't require any additional hardware or device. All interaction between the user and the Color LCD are performed directly with the display as the interactive device using the pop-up touchscreen keypad. This configuration would be hard to use due to the limited number of characters that can be displayed or if you're developing a graphics program.

Configuring the Color LCD Display

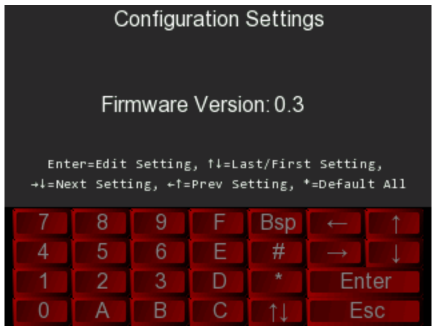
To perform the exercises outlined in this manual, the Color LCD must be configured to operate in ACS Basic mode. This mode is set by changing a configuration item using the touchscreen on the display.

On reset or power-up the Color LCD looks for a constant touch in one of four places. If it sees the touchscreen being pressed as it comes up then it will bypass normal operation and enter one of four different screens. The screen that we want is the Configuration Settings.



Press and hold the lower-left quadrant of the screen. Then either power-up the Color LCD or reset it using the RESET push button on the back.

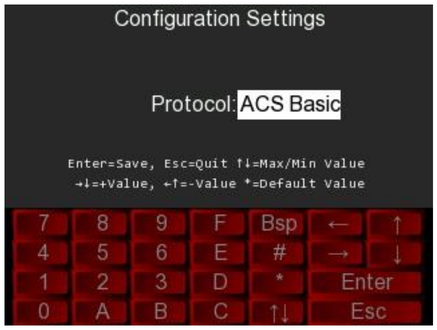
After the display performs its power-up initialization the Configuration Settings screen should be displayed:


Configuration Settings

Firmware Version: 0.3

Enter=Edit Setting, ↑↓=Last/First Setting,
→↓=Next Setting, ←↑=Prev Setting, *=Default All

Press the on-screen keypad down arrow once to select the Protocol: setting. Pressing the Enter key will open the setting for editing. Use the up/down arrows to change the setting to ACS Basic then press Enter again.


Configuration Settings

Protocol: ACS Basic

Enter=Save, Esc=Quit ↑↓=Max/Min Value
→↓=+Value, ←↑=-Value *=Default Value

There’s at least one more setting to configure. If you’re communicating using the display SERIAL port, you should configure a matching baud rate that matches your attached communication device. The display defaults to 9600 baud, 8 data bits, 1 stop bit and no parity. Pressing the on-screen keypad down arrow once more should show the Serial Baud Rate: setting. You can edit this in the same way to configure your desired baud rate. Successive down arrows should advance you through the Serial Data Bits, Serial Stop Bits and Serial Parity settings. You should configure your terminal emulator with serial settings that matches your display’s configuration. The ColorLCD defaults to 9600 baud, 8 data bits, 1 stop bit and no parity. Start Hyperterminal, Tera Term VT, PuTTY or other terminal emulator program on your connected computer (PC). Establish a connection to the connected serial port. Power-up or reset the ColorLCD.

If you’ve configured the ColorLCD USB port to act as a serial communication device the baud rate does not matter. You must have installed the ColorLCD.inf driver when the unit was first connected to the PC. Powering or resetting the display will disconnect and reconnect the USB requiring a re-establishing of the connection in your terminal emulator.

If you're communicating using the Ethernet and TCP/IP Raw Sockets you should configure the terminal emulator connection to use the IP address and port number that matches your display's configuration. The ColorLCD defaults to an IP address of 192.168.1.200 with a raw socket port of 23. Powering or resetting the ColorLCD will disconnect from the terminal emulator requiring a re-establishing of the connection.

Start Hyperterminal, Tera Term VT, PuTTY or other terminal emulator program on your connected computer (PC) establishing the connection as outlined above.

Now power-up or reset the Color LCD Display. This message should appear on both the PC's terminal emulator program and the display:

```
ACS Basic v3.0 May 12 2014 10:49:30
Ready
```

(the v3.0 and date/time specify which version Basic the Color LCD Display is running)

If you don't receive this message, check your power supply and wiring. Try turning the Display off and on again. Remember if connecting via USB or Ethernet you must re-establish the connection each time the Display is power-cycled or reset.

Once you see the message, press the Enter key. You should see an additional Ready prompt.

Once you see this message on your PC and Basic receives your key presses you're ready to begin.

What the Color LCD Understands

In this beginners section of the manual you will learn how to talk to the Color LCD. In computer terminology this is referred to as ***programming***. Once you learn how to program you can get your display to do whatever you tell it to do – usually.

The Color LCD understands a language called ACS Basic. ACS Basic is a customized form of the "Beginners All-purpose Symbolic Instruction Code". The original Basic was developed back in 1964 to provide computer access to people who didn't usually program. This version provides additional commands and features that make it useful to program an interactive display.

Talking to the Color LCD

Try pressing the Enter↵ key on your PC's keyboard. The Color LCD display indicates that it is awaiting instruction by responding with Ready. Ready is the display's prompt – it is waiting for a command from you.

Try typing the following as your first command – type this exactly as it is shown:

```
PRINT “Hello World”
```

When you reach the end of the line, review it for mistakes. Did you put the quotation marks where they were shown? If you made a mistake, simply press the backspace key and the last character that you typed will disappear. You can backspace over the entire line if necessary.

Your PC (and display screen) should look like this:

```
Ready
PRINT "Hello World!"
```

Now press the Enter↵ key and see what happens. Your screen should now look like this:

```
Ready
PRINT "Hello World!"
Hello World!
Ready
```

The Color LCD display executed the command that you entered by printing the message that you had in quotes.

Now let's try another command:

```
PRINT “2 + 2” Enter↵
```

The Color LCD executes your command by printing:

```
2 + 2
Ready
```

If you expected to see the number four, then try removing the quotation marks:

```
PRINT 2 + 2 Enter↵
```

The Color LCD executes your command by printing:

```
4
Ready
```

The Color LCD sees everything that you type as either Strings or Numbers. If it's in quotes, it's a String and the display sees it exactly as it was typed. If it's not in quotes it's a Number. The Color LCD will figure it out like a numerical problem. It it's not in quotes the display can interpret it by adding, subtracting, multiplying or dividing it.

Let's try a multiplication problem:

```
PRINT 1589 * 2 Enter
```

```
PRINT 1589 * 2
3178
Ready
```

The Color LCD uses an asterisk as a multiplication sign rather than an X since the X character is alphabetical and can be part of a name as we will see later.

There Are Rules…

The Color LCD is very literal. If it doesn't understand what you have typed it will produce an error message to let you know. Try typing this line deliberately misspelling the word PRINT:

```
PRIINT "HI" Enter
```

The Color LCD prints:

```
PRIINT "HI"
Syntax error - no equals
Ready
```

The display doesn't understand what you have typed. The "Syntax error" message indicates that the command "PRIINT" is not one that it knows how to do. It also tries to provide additional information to help you figure out the error, "- no equals", but in this case it is confused and this doesn't help.

Try leaving off the last quotation mark. Type:

```
PRINT "HI Enter
```

The Color LCD prints:

```
PRINT "HI
Mis-matched quotes error
Ready
```

The Color LCD will also give you error messages when it does understand what you have typed, but the command will result in incorrect operation. For instance, try typing:

```
PRINT 5 / 0 Enter
```

The display prints:

```
PRINT 5 / 0
Divide by zero error
Ready
```

This error message indicates that you're asking it to divide by zero – which is not possible.

Remembering Numbers and Strings

One of the features of the Color LCD is the ability to remember things you ask it to. For example, to make the display remember the number 13, type this:

```
A = 13 Enter↵
```

Now try the other PRINT commands that you did before. To see if the display still remembers what A is equal to, type:

```
PRINT A Enter↵
```

And the display should print:

```
PRINT A
13
Ready
```

The Color LCD remembers that the placeholder A has the number value of 13 until you turn it off or change it. Type:

```
A = 17 Enter↵
```

```
A = 17
Ready
PRINT A
17
Ready
```

So what happened? When you first set A equal to 13 the display created a named memory location to hold that value. You can refer to that location by the name A to retrieve the current value of the memory location as with the PRINT A command above. You can also modify the value of the named memory location by setting it to a new value. You can use combinations of letters and numbers to name these memory locations – the only requirement is that the name must not start with a number. Try typing these commands:

```
B = 15 Enter↵

C2 = 20 Enter↵

LongName = 25 Enter↵
```

Now ask the Color LCD to retrieve all of these numbers by name. Type:

```
PRINT A, B, C2, LongName Enter↵
```

And the display should show:

```
PRINT A, B, C2, LongName
17 15 20 25
Ready
```

To get the Color LCD to remember strings of letters or numbers, put a dollar sign at the end of the name. Type:

```
A$ = "Remember" Enter↵

B$ = "this for me" Enter↵
```

And then ask the display to display them:

```
PRINT A$, B$
Remember this for me
Ready
```

In computer terminology these named memory locations are referred to as ***variables***. The Color LCD keeps track of these variables and their current values for you. You can ask it for a list of what variables it is currently remembering by typing the command:

```
VARS Enter←
```

And the display will show the variables that we've used so far along with their current contents:

```
VARS
A                                R/W Int         = 17
B                                R/W Int         = 15
C2                               R/W Int         = 20
LongName                         R/W Int         = 25
A$                               R/W Str$        = "Remember "
B$                               R/W Str$        = "this for me"
Ready
```

The **R/W** indicates that the variable can be both **R**ead from and **W**ritten to. Later on we will see how to make variables **R**ead **O**nly. The **Int** indicates that the display understands this variable to be for holding Numbers, **Str$** indicates that the variable holds strings.

And of course the Color LCD is picky about what is stored in the two types of variables. Try typing these lines:

```
D = "6" Enter←
D$ = 6 Enter←
```

With both of these lines the display responds with an error message:

```
D = "6"
Wrong expression type error - can't assign string to numeric var
Ready
D$ = 6
Wrong expression type error - can't assign number to string var
Ready
```

In computer terminology, setting variables to values is referred to as ***assigning a value to a variable***.

There are four fundamental rules for variable value assignment:

Rules for Numeric Data

1. Numbers not in quotes are Numeric Data
2. Numeric Data can only be assigned to variables named without a trailing dollar sign

Rules for String Data

1. Any data in quotes is String Data
2. String Data may only be assigned to variable name with a trailing dollar sign

Variable Rules

1. You may use multiple characters from the upper case letters A-Z, the lower-case letters a-z, the numbers 0-9 and the '_' underscore character for variable names.
2. The first character of the name must be a letter, not a number or underscore.
3. Variable names are case-sensitive: **Aname** is not the same as **aNAME**.
4. Variables whose names end with a dollar sign can only hold String Data, otherwise they can only hold Numeric Data.
5. The list of current variables can be shown with the VARS command.
6. Short, concise variable names take less memory and work slightly faster.

Remembering Commands

First we need to erase everything that the Color LCD has remembered so far. Type:

```
NEW Enter←┘
```

Now type this command line: Be sure and type the number 10 first:

```
10 PRINT “Hello from the Color LCD” Enter←┘
```

Notice that this time, when you pressed Enter←┘ nothing appeared to have happened. Not that you can immediately see. What you did was to type your first program. Type:

```
RUN Enter←┘
```

The display now runs your program. You can type RUN again and again. You can also type run – ACS Basic’s commands are not case-sensitive, just variable names:

```
10 PRINT "Hello from the Color LCD"
RUN
Hello from the Color LCD
Ready
run
Hello from the Color LCD
Ready
```

So the number at the beginning of the line tells Basic to store the line instead of executing it. Let’s add another line to the program. The number at the beginning of the line is the “line number”. Type:

```
20 print “What’s your name?” Enter←┘
```

Now let’s ask the Color LCD to show us the entire program. Type:

```
LIST Enter←┘
```

The display LISTs your entire program so far – notice how the lower-case print command was capitalized and how the lines are stored and shown in line number order:

```
LIST
10 PRINT "Hello from the Color LCD"
20 PRINT "What's your name?"
Ready
```

Now run the program: Type: RUN Enter←┘ The Color LCD prints:

```
RUN
Hello from the Color LCD
What's your name?
Ready
```

Try answering the question by typing your name and pressing Enter←┘ . . . there’s that Syntax error again – the display didn’t understand what you meant when you typed in your name:

```
RUN
Hello from the Color LCD
What's your name?
Ready
Steve
Syntax error - no equals
Ready
```

In fact you have to instruct the Color LCD to accept your answer by giving it a command, INPUT, to do so. Add this line to the program:

```
30 INPUT Name$ Enter←┘
```

The INPUT command tells the display to stop and wait for you to type something which it will assign to the variable Name$. Add one more line to the program to show the Name that was entered:

```
    40 PRINT “Hi, “, Name$ Enter↵
```

Now LIST the program. It should look like:

```
LIST
10 PRINT "Hello from the Color LCD"
20 PRINT "What's your name?"
30 INPUT Name$
40 PRINT "Hi,", Name$
Ready
```

Now let’s RUN the program:

```
RUN
Hello from the Color LCD
What's your name?
? Steve
Hi, Steve
Ready
```

You can run the program many times, answering the question with different names – the display doesn’t care what name you use. After each RUN the variable Name$ holds the last name you entered.

You can make your program run over and over without having to type the RUN command each time. Add this line to the program:

```
    50 GOTO 10 Enter↵
```

Now RUN the program. It runs over and over without stopping. The GOTO command told the Color LCD to go back up to line 10. Your program will repeat over and over because every time it executes line 50 it jumps back to line 10. In computer terminology this is referred to as a ***loop***. The only way to stop this endless loop is to press the esc key twice in a row followed by the Enter↵ key. This is known as ESCaping the program and the display tells you that you escaped:

```
Hello from the Color LCD
What's your name?
? esc esc Enter↵
 ESC at line 30
Ready
```

In this program the Enter↵ key was required following the double esc key because the Color LCD was waiting for the Enter↵ key at the INPUT command. If the display is not waiting for INPUT the double esc key is sufficient to ESCape the program.

Changing your Program

So how can you modify your program without typing NEW and starting over each time? To replace an existing program line simply type the new line using the same line number at the beginning of the line you want to replace. Type:

```
50 GOTO 40 Enter←
```

Your program should now look like:

```
50 GOTO 40
LIST
10 PRINT "Hello from the Color LCD"
20 PRINT "What's your name?"
30 INPUT Name$
40 PRINT "Hi,", Name$
50 GOTO 40
Ready
```

This program change modifies the loop to not ask the question over and over. Instead the program now just keeps repeatedly PRINTing the Name$ that you INPUT the first time. Press the esc key twice in a row to ESCape the program when you've seen enough. Notice that you don't need to press the Enter← key to ESCape this time because the display is not waiting for INPUT:

```
RUN
Hello from the Color LCD
What's your name?
? Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve
Hi, Steve esc esc
 ESC at line 40
Ready
```

To remove a line from your program simply type the line number followed by the Enter← key. This erases that line from your program. Type:

```
50 Enter←
```

This removes line 50 from your program:

```
50
LIST
10 PRINT "Hello from the Color LCD"
20 PRINT "What's your name?"
30 INPUT Name$
40 PRINT "Hi,", Name$
Ready
```

Put line 50 back into your program. Type:

```
50 GOTO 40 Enter←
```

Now let's change the way that the PRINT shows your name. Replace line 40 in your program by typing it again, but add a semicolon at the end:

```
40 PRINT "Hi,", Name$; Enter←
```

Now RUN the program. Notice how the trailing semicolon crams everything together?

```
40 PRINT "Hi, ", Name$,
LIST
10 PRINT "Hello from the Color LCD"
20 PRINT "What's your name?"
30 INPUT Name$
40 PRINT "Hi, ", Name$,
50 GOTO 40
Ready
run
Hello from the Color LCD
What's your name?
? Steve
Hi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi, SteveHi,
SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,
SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,
SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,
SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,
SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,  SteveHi,
SteveHi, Ste ESC at line 40
Ready
```

Controlling the Program Execution

You've seen that a program can be RUN from lowest line number to highest. The program can also loop using the GOTO command. And a running program may be ESCaped to stop.

Another way to control the execution of a program is to GOTO only if a certain condition is met. Let's change our program so that it stops if we enter a blank name. Modify line 40 to remove the trailing comma by typing:

```
40 PRINT “Hi,“, Name$ Enter←
```

Now replace line 50 with a new IF / THEN command. The new IF / THEN command evaluates a condition, and, IF the condition is True, THEN it executes the command after the THEN. Otherwise execution continues with the following program line. Type:

```
50 IF Name$ <> “” THEN GOTO 20 Enter←
```

In this example the IF condition that is being evaluated is: **Name$ < > “”**. Name$ is the variable that receives your INPUT up until the Enter← key. The < > (less than, greater than) stands for “not equal to”, and the “” is an empty string – a string with no characters in it. So your program should now look like:

```
LIST
10 PRINT "Hello from the Color LCD"
20 PRINT "What's your name?"
30 INPUT Name$
40 PRINT "Hi,", Name$
50 IF Name$ <> "" THEN GOTO 20
Ready
```

Run the program. Try entering your name the first time your program asks for it, then the next time simply press the Enter← key. Notice how the program loops back to line 20 when your name was INPUT otherwise the program stops:

```
RUN
Hello from the Color LCD
What's your name?
? Steve Enter←
Hi, Steve
What's your name?
? Enter←
Hi,
Ready
```

When you press the Enter key in response to the INPUT command without typing any characters first, the INPUT variable receives an empty string otherwise the variable holds any characters that were typed in before the Enter key.

You can think of line 50 as reading: IF variable Name$ not equal to empty string THEN GOTO line 20. And that is exactly what the program is doing. However there is an error. In computer terminology this is called a ***program bug***. Notice how the program still prints Hi, before it stops when nothing is ENTERed. This is because the program executes line 40 before it checks for Name$ being empty. In order to fix this 'bug' we have to check for the stop condition before we PRINT the Name$.

Replace line 40 with the new IF / THEN check. Type:

```
40 IF Name$ = “” THEN STOP Enter
```

And then replace line 50 with the PRINT command. Type:

```
50 PRINT “Hi,“, Name$ Enter
```

The program should now look like:

```
LIST
10 PRINT "Hello from the Color LCD"
20 PRINT "What's your name?"
30 INPUT Name$
40 IF Name$ = "" THEN STOP
50 PRINT "Hi,", Name$
60 GOTO 20
Ready
RUN
Hello from the Color LCD
What's your name?
? Steve
Hi, Steve
What's your name?
?
STOP in line 40
Ready
```

The new STOP command does exactly that – it stops the program from running and shows you what line the program stopped on. You can also use the END command which stops without the message. Try it by changing line 40.

Program Rules

1. A program consists of one or more command lines that begin with a line number.
2. Commands are not case-sensitive but are converted by Basic. Variables are case-sensitive.
3. Program lines and variables may be cleared from memory by the NEW command.
4. Program lines are kept in ascending numeric order by the display.
5. A program is RUN to execute the lines in numeric order starting with the lowest numbered line.
6. Program lines may be shown with the LIST command.
7. The execution order of program lines may be changed by the GOTO command.
8. A program which is executing repeatedly in a loop may be ESCaped by pressing the esc key twice in a row.
9. A program line may be replaced by typing a new program line with the same line number.
10. A program line may be deleted by typing the line number only followed by Enter.
11. Program line execution may be conditioned by the IF / THEN command.
12. Program execution may be stopped with the STOP or END commands.

Learning How to Count

Most programs require the ability to count – lines, key presses, characters, loops or other things. With the commands that you already know you can write a program that counts.

First we need to learn a fundamental concept. In order to add to a variable you can use it on each side of the assignment equals sign. For example, type:

`A = 1` Enter↵

`PRINT A` Enter↵

`A = A + 1` Enter↵

`PRINT A` Enter↵

With assignment A = A + 1, the current value of A has one added to it then it becomes the new value of A. The value to add doesn't have to be 1, it can also be the value of another variable. You can read this statement as "Variable A equals the value of Variable A plus 1".

So in order to count from 1 to 10 we can write the following short program. Start fresh by typing:

`NEW` Enter↵

First we set the value of the counter, A, equal to 1. Type:

`10 A = 1` Enter↵

Then we print it. Type:

`20 PRINT A` Enter↵

Then we add one to it. Type:

`30 A = A + 1` Enter↵

Then we check to see if A is less than or equal to 10… and, if so, we loop back to print the new value. Otherwise we're done counting:

`40 IF A <= 10 THEN GOTO 20` Enter↵

Now list the program. It should look like:

```
LIST
10 A = 1
20 PRINT A
30 A = A + 1
40 IF A <= 10 THEN GOTO 20
Ready
```

Now RUN it:

```
RUN
1
2
3
4
5
6
7
8
9
10
Ready
```

Simple – right? However, ACS Basic provides another way to count with fewer commands that's a little easier to read.

Start fresh by typing NEW again. Now we will work with a new two command combination: FOR / NEXT.

The FOR command replaces two commands: line 10 - the setting of the initial count value, and line 40 – the check for the end count value. The NEXT command replaces two commands: line 30 – the advance of the count value, and the GOTO portion of line 40 – the counting loop. Let's see how this works.

First the FOR command. This command determines the count variable, its initial value and the ending value. Type:

```
10 FOR A = 1 TO 10
```

Now as before we PRINT the count value. Type:

```
20 PRINT A
```

And finally the NEXT command. This command advances the count value and executes the loop if more counting is necessary. This is referred to as closing the loop. Type:

```
30 NEXT A
```

Now list the program. It should look like:

```
LIST
10 FOR A = 1 TO 10
20 PRINT A
30 NEXT A
Ready
```

Now RUN it:

```
RUN
1
2
3
4
5
6
7
8
9
10
Ready
```

This program is one line shorter, but does the same thing. It is also easier to read and comprehend. Try changing the initial and ending value to see the results.

Great! But what if we want to count by two – how do we do that. It seems like the A = A + 1 is implied.

The Basic language designers thought of that and have provided another keyword that can be added to the end of the FOR command to specify the value to be added each time around the loop – STEP.

The STEP keyword goes at the end of the FOR command – after the ending value. Let's replace line 10 with this new FOR command that has a STEP 2 on the end:

```
10 FOR A = 1 TO 10 STEP 2
```

Your program should now look like:

```
LIST
10 FOR A = 1 TO 10 STEP 2
20 PRINT A
30 NEXT A
Ready
```

Now RUN it:

```
RUN
1
3
```

```
5
7
9
Ready
```

The initial value, ending value and optional STEP value don’t have to be numbers – they can also be numeric variables. And the STEP value can be negative to. Let’s replace line 10 again to try this. Type:

```
10 FOR A = 10 TO 1 STEP -1 Enter↵
```

Your program should now look like:

```
LIST
10 FOR A = 10 TO 1 STEP -1
20 PRINT A
30 NEXT A
Ready
```

and when you RUN it:

```
RUN
10
9
8
7
6
5
4
3
2
1
Ready
```

There’s a few more important things to know about FOR / NEXT. First, you shouldn’t try to GOTO into the middle of a FOR / NEXT loop. When the Color LCD encounters the NEXT command without having seen the matching FOR first it gets confused and gives you an error. Here’s an example of that error – we add a line at the beginning of the program to GOTO line 20 which is inside the FOR / NEXT loop:

```
LIST
5 GOTO 20
10 FOR A = 10 TO 1 STEP -1
20 PRINT A
30 NEXT A
Ready
RUN
 0
Nesting error in line 30 - NEXT without preceding FOR
```

The second thing to remember is that if you nest FOR / NEXT loops – one inside of another, you must close the inner loop before closing the outer loop. Here’s the right way to nest loops:

```
LIST
10 FOR X = 1 TO 3
20 FOR Y = 3 TO 1 STEP -1
30 PRINT "X = ",X, " Y = ",Y
40 NEXT Y
50 NEXT X
Ready
RUN
X =  1 Y =  3
X =  1 Y =  2
X =  1 Y =  1
X =  2 Y =  3
X =  2 Y =  2
X =  2 Y =  1
X =  3 Y =  3
X =  3 Y =  2
X =  3 Y =  1
Ready
```

And here’s the wrong way to nest loops:

```
LIST
10 FOR X = 1 TO 3
20 FOR Y = 3 TO 1 STEP -1
30 PRINT "X = ",X, " Y = ",Y
40 NEXT X
50 NEXT Y
Ready
RUN
X =  1 Y =  3
Nesting error in line 40 - NEXT var doesn't match FOR var
Ready
```

The third thing to remember is that you shouldn't try to GOTO out of a FOR / NEXT loop. While you won't receive an immediate error, the cleanup performed by the NEXT statement isn't performed and if you do it enough times you will get an error when there isn't sufficient memory left for the display to remember more FOR / NEXT commands. Exiting a FOR / NEXT loop early is a fairly common programming requirement however so ACS Basic provides a system form of GOTO to jump out – EXITFOR. To try it, modify our existing program. First remove line 5 by typing:

`5` Enter←

And then add line 40 to indicate that we're done counting and provide a target line for the EXITFOR command:

`40 PRINT “Done”` Enter←

And finally add a condition check for exiting the FOR / NEXT loop – let's say when the count variable is equal to 5 we want to leave the loop. Type:

`15 IF A = 5 THEN EXITFOR 40` Enter←

Your program should now look like:

```
LIST
10 FOR A = 10 TO 1 STEP -1
15 IF A = 5 THEN EXITFOR 40
20 PRINT A
30 NEXT A
40 PRINT "Done"
Ready
```

And when you RUN it:

```
RUN
10
9
8
7
6
Done
Ready
```

Counting Rules

1. A variable can appear on both sides of the assignment equal sign. The variable on the right hand side refers to its existing value and the left hand side variable will receive the new value.
2. You can implement a count sequence by assigning an initial value to a variable, advancing it by adding or subtracting to it then checking if it is at the ending value and looping back if it is not.
3. Basic provides a FOR / NEXT command with an optional STEP clause to simplify counting.
4. Nested FOR / NEXT loops must be closed from the inside out – inside loops first.
5. You can't jump into the middle of a FOR / NEXT loop – an error will result on the NEXT command.
6. You should only jump out of a FOR / NEXT loop using the system EXITFOR command.

Remembering your Programs

The Color LCD wouldn’t be very useful if you always had to type in your programs every time you turned it on. The longer your programs are the more of a problem this would become.

The Color LCD has a slot for a micro SD memory card. You can SAVE your programs by name on the card and LOAD or RUN them later by referring to the same name.

Think of the micro SD card as a file drawer that holds many files. You can create a new file and put it in the drawer, or retrieve an existing file from the drawer. You can’t retrieve a file that hasn’t been created first. The Color LCD will keep a directory listing of what files are in the drawer (on the micro SD card).

Let’s try it. Put a fresh card into the slot and ask for a directory of files. Type:

```
DIR Enter
```

On a fresh card you should see the following:

```
DIR
---------------------------------------------
                                             0 files
                                             0 directories
Ready
```

The display is showing you that there are currently no files on the card. Now type in one of your earlier programs – remember to start with NEW Enter. Let’s start with the simple FOR / NEXT program:

```
NEW
10 FOR A = 1 TO 10
20 PRINT A
30 NEXT A
```

You can RUN it to verify that it works as before. Now let’s SAVE it in a file. Let’s call it FORNEXT. Type:

```
SAVE FORNEXT Enter
```

Did it save it? Let’s ask for a directory again:

```
DIR
FORNEXT.BAS                                 43 A      02-26-2012 04:57:02 PM
---------------------------------------------
                                             1 files
                                             0 directories
Ready
```

There it is. The display is telling us that it’s 43 characters long, it was created on 2-26-2012 at 4:57PM and that there is one file on the card. What’s the .BAS on the end of the file name? There can be many types of files in the drawer (on the card) and this is Basic’s way of telling us that this file is a Basic program. Great! Now let’s try to get it back.

First clear out the program and variables with NEW. Try a LIST and VARS to see that there’s actually nothing in memory:

```
NEW
Ready
LIST
Ready
VARS
Ready
```

Now let’s retrieve it. Type:

```
LOAD FORNEXT Enter
```

and then LIST it:

```
LOAD FORNEXT
Ready
```

```
LIST
10 FOR A = 1 TO 10
20 PRINT A
30 NEXT A
Ready
```

and there it is. You can RUN it to verify that it still works.

Modify it to count backwards by changing line 10 and adding the STEP keyword:

```
10 FOR A = 10 TO 1 STEP -1 Enter←
```

You can LIST and RUN it. Then let's SAVE this as FORNEXTSTEP. Type:

```
SAVE FORNEXTSTEP Enter←
```

Now let's ask for a directory again:

```
DIR
FORNEXT.BAS                                 43 A      02-26-2012 04:57:02 PM
FORNEXTSTEP.BAS                             51 A      02-26-2012 05:12:56 PM
---------------------------------------------
                                             2 files
                                             0 directories
Ready
```

Both programs (files) are there. ACS Basic provides a shortcut for loading and running a program file – just type RUN followed by the file name. Basic will do the NEW, LOAD and RUN all in a single step:

```
RUN FORNEXT
1
2
3
4
5
6
7
8
9
10
Ready
RUN FORNEXTSTEP
10
9
8
7
6
5
4
3
2
1
Ready
```

Saving Rules

1. Programs can be filed onto a micro SD card installed in the Color LCD.
2. A directory of what files are on the card can be obtained by the DIR command.
3. Programs are saved to the card with the SAVE command.
4. Programs are retrieved from the card with the LOAD command.
5. Programs may be retrieved and executed with the RUN command.
6. ACS Basic remembers the last program file name used with LOAD, RUN or SAVE so a modified program can be saved into the same file using a SAVE command without a name.

Things to do with Numbers

So far we've seen that the Color LCD can add and multiply numbers – using numbers directly or numbers stored in variables. The display can also subtract and divide numbers. In computer terminology these actions are referred to as ***operators***. Let's try a few:

```
PRINT 4 - 2
2
Ready
PRINT 9 / 3
3
Ready
```

Here's a new operator that you probably haven't heard of or used before – remainder of division. In computer terminology this is referred to as a ***modulo operation***. The first number is divided by the second number and the remainder is returned. So if two numbers divide evenly, the modulo is zero – no remainder:

```
PRINT 10 % 5
0
Ready
```

However if the two numbers don't divide evenly, the modulo operator returns the remainder after the division that would be performed. So if we divide 5 by 4, the remainder or modulo would be 1:

```
PRINT 5 % 4
1
Ready
```

Operators usually go between two numbers or numeric variables. However there are some operators that go in front of a single number or variable. In computer terminology these are referred to as ***unary operators***. There are a couple of these, but the most common is a leading minus sign – referred to as negate:

```
A = 10
Ready
PRINT -A
-10
Ready
```

This is just a shorthand way of telling the Color LCD to subtract the number from zero.

Only Whole Numbers Please

Now let's try dividing two numbers that don't divide evenly – say 10 divided by 3:

```
PRINT 10 / 3
3
Ready
```

What happened? Shouldn't 10 divided by 3 equal 3 and 1/3? It should but this shows a limitation of ACS Basic – it only knows how to work with whole numbers – not decimals or fractions. So when it divides 10 by 3 it returns the whole number part and discards the fractional part. In computer terminology this is referred to as ***integer arithmetic***. While this can be a limitation for solving some problems most applications of the Color LCD can be performed only using integers – and there are some workarounds that we'll examine later. Remember, the modulo operator can return the remainder of the same division.

The Size of Numbers

So if ACS Basic can only work with whole numbers, how big can they be? What's the limit? This Basic works with what programmers refer to as short integers – they are 32 computer bits wide. A computer bit can only be on or off – when you have 32 of them across, with each one able to be on or off, the biggest number that they can represent is +4,294,967,295. which is the number 2 raised to the 32nd power (4,294,967,296) less one reserved for the zero = 4,294,967,295.

Actually the computer in the ColorLCD reserves 1 bit for a plus or minus indicator – a sign bit. So there are really only 31 bits available for the actual number. So the maximum positive number is +2,147,483,647 and the maximum negative number is –2,147,483,648.

Comparing Numbers

There are also operators for comparing two numbers. Numbers can be checked to see if they're the same or not or if one is larger or smaller than another. In computer terminology these checks are called ***comparison operators*** – two numbers are compared with each other. The results of a comparison are either true or false. In ACS Basic, a true comparison results in a 1 and a false comparison results in a zero.

To check for equality (two numbers being the same) we use the equal operator – which is the equals sign:

```
PRINT 1 = 0
0
Ready
PRINT 2 = 2
1
Ready
```

So how do you test if two numbers are not equal? ACS Basic uses a less than followed immediately by a greater than (no space in-between) as the not equal operator. Notice how this check gives the opposite result of the equal check.

```
PRINT 1 <> 0
1
Ready
PRINT 2 <> 2
0
Ready
```

Of course we can also check for numbers being less than, less than or equal, greater than and greater than or equal:

```
PRINT 2 < 2
0
Ready
PRINT 2 <= 2
1
Ready
PRINT 4 > 4
0
Ready
PRINT 4 >= 4
1
Ready
```

Combining Numbers in Order

You don't have to only do a single operation on numbers at a time. You can combine them into a series of operations to achieve the result you want. In computer terminology this is referred to as an ***expression***.

Usually, when you perform a bunch of operations on numbers you do them in a left-to-right order. So if you want to add two numbers, and now double the result you might write:

```
PRINT 2 + 3 * 2
8
Ready
```

You would expect the result to be 2 plus 3 equals 5 times 2 equals 10 – right? How did the Color LCD come up with the number 8?

It turns out that in Basic, like most computer languages there is an order in which numeric operations are performed. In computer terminology this is referred to as ***operator precedence*** or ***operator priority***. If there wasn't an defined order then you could get different results each time or perhaps different results between different machines. Negation is performed first, then Multiplication, Division and Modulo, then Addition and Subtraction, then Comparisons are performed last. So the display does the 3 times 2 first, then it adds the 2 which gives 8 instead of 10.

What if you really want these operations done in the order that you wrote them down? Well it turns out that you can specify the order by grouping operations together using parenthesis. The Color LCD Basic will execute operations inside parenthesis first, then use that result to perform the next operation in the expression. So to get the result that you wanted above, you could write:

```
PRINT (2 + 3) * 2
10
Ready
```

So when in doubt, or when you really want things done in a certain order, use parenthesis to group operations together. The display will evaluate things from the inside out starting with the most nested set of parenthesis first.

Numeric Operator Rules

1. Numbers can be added, subtracted, multiplied, divided, remaindered, negated and compared.
2. Multiple operations can be performed sequentially to form a numeric expression.
3. Multiple operations are performed in a priority fashion from first to last: negation, multiplication and division, addition and subtraction, then comparison.
4. Multiple operations with the same priority are performed left to right.
5. Parenthesis can be used to change the order in which numeric expressions are evaluated.

Things to do with Strings

There are also operations that can be performed on strings. The most common of these are to combine two separate strings into a new single string by ‘adding’ one string onto the end of another. In computer terminology this is referred to as ***concatenation*** – the two strings are concatenated together:

```
PRINT "HELP" + "ME"
HELPME
Ready
Part1$ = "Help"
Ready
Part2$ = "Me"
Ready
PRINT Part1$ + " " + Part2$
Help Me
Ready
```

Comparing Strings

Like numbers you can also compare strings. The two strings are compared, a character at a time to determine if they are the same, less than or greater than each other:

```
PRINT "ONE" = "TWO"
0
Ready
PRINT "ONE" <> "TWO"
1
Ready
PRINT "ONE" > "TWO"
0
Ready
PRINT "ONE" < "TWO"
1
Ready
```

Now for Something at Random

By now we've learned a few different commands that the Color LCD Basic understands. Let's try something new. Type this line:

```
10 PRINT RND(10) Enter
```

And now RUN it. The display printed a random number between 0 and 9. RUN it a few more times:

```
10 PRINT RND(10)
Ready
RUN
8
Ready
RUN
7
Ready
RUN
5
Ready
RUN
1
Ready
```

Now make the program loop back to run continuously. Add line 20 to the program and RUN it. You will have to ESCape the program to stop it:

```
20 GOTO 10
LIST
10 PRINT RND(10)
20 GOTO 10
Ready
RUN
7
8
6
9
8
0
0
7
4
9
9
 ESC at line 10
Ready
```

What if we want to have random number from 0 to 100? Changed line 10 to this and RUN it:

```
10 PRINT “”, RND(100);
LIST
10 PRINT “”, RND(100);
20 GOTO 10
Ready
RUN
 59 51 31 73 82 62 38 92 78 58 71 84 28 94 46 43 2 59 26 80 32 41 44 43 12 31 30 84 18 58 4 80
11 3 56 39 14 60 0 42 33 95 95 9 56 80 91 16 54 7 86 37 12 25 45 37 82 67 29 95 26 9 23 62 74 57
80 6 16 83 85 68 64 39 22 45 37 81 50 56 93 56 58 4 70 17 0 28 81 99 61 22 36 74 27 9 19 12 9 38
32 87 41 18 35 12 70 28 40 60 22 38 83 23 58 23 17 30 75 95 11 46 12 67 7 26 52 7 39 1 70 67 69
39 53 0 63 28 25 94 39 27 86 94 7 20 25 52 94 88 45 38 99 25 92 72 25 45 15 67 5 48 19 66 71 20
51 74 12 36 89 24 6 4 60 80 29 55 65 46 48 30 29 14 86 80 7 84 50 26 74 53 78 44 3 9 19 94 8 24
42 99 92 58 31 73 27 46 23 55 49 29 82 27 75 49 97 54 75 64 88 62 49 8 65 52 73 36 9 74 60 35 62
37 46 63 1 88 46 50 64 5 96 21 43 68 19 77 51 37 71 57 25 55 65 21 84 89 77 72 72 76 68 84 49 6
93 82 42 93 57 75 39 87 94 70 94 61 97 32 45 79 39 92 24 10 49 20 8 7 39 7 16 78 2 16 15 90 53
19 10 11 76 82 99 70 27 83 97 94 29 34 78 35 69 22 70 26 29 84 90 97 97 11 16 3 42 83 77 52 86
39 48 91 80 37  ESC at line 10
Ready
```

It's a Function!

So the function of **RND( )** is to produce random numbers. It takes a numeric value between the parentheses, computes a random number based upon the number that it's given and then takes on the value of that random number. It's kind of like a command and kind of like a variable at the same time. In computer terminology these combination command / variables are referred to as ***functions***. Because they are part of the Basic language and don't have to be rewritten each time you want to use them they are referred to as ***built-in functions***.

Since functions act like a variable you can use them wherever you would read a variable – like in a PRINT command, an IF / THEN conditional check or in an expression. The difference between a function and a variable is that you can't assign a value to a function – you can only pass arguments to it and then use the value that it assumes as a result.

In computer terminology this is referred to as ***calling a function that takes an argument and returns a value***. It's like a variable, but you don't assign values to it – instead you pass it one or more arguments between the parenthesis and it acts like a variable whose value changes based upon the arguments.

There are other functions that are very useful when writing Basic programs. Some take no arguments but return a value – either numeric or string. Some take one or more numeric arguments and return a numeric value. Some take string arguments and return a numeric value. Some take both string and numeric arguments and return a value.

So how do you know what type of value a built-in function will return? Just like a Basic variable – it's all in the name. If a built-in function returns a string value, the function name ends with a dollar sign – otherwise it returns a numeric value. This makes sense because functions can be used in place of variables.

Function Rules

1. Functions behave like read-only variables that can supply different values or behavior in your programs depending upon what they are pre-defined to do and what arguments that they may require to do it.
2. When calling a function, the function name must be immediately followed by the opening parenthesis surrounding the function's arguments.
3. The type of a function is identified by the name, just like a variable.

A Casual Remark

The REM command allows you to put notes (REMarks) into your programs. The REM command actually does nothing - when ACS Basic 'sees' the REM command it skips over the rest of the line. In computer terminology the use of the REM command is referred to as ***commenting your code***. It serves a couple of purposes – reminding yourself about what you were trying to do with this code, and documenting what you hope that it does as an aid for others. While it doesn't seem important now with these tiny programs, it will become increasingly important when the programs get larger and you have many more of them to keep track of. It's considered good practice.

```
10 REM This is a comment line
```

Let's examine some of these functions. We will group them by what type of value they will return – numeric or string.

Functions Returning a Number

You've already seen one of these type of functions – **RND( )**. Remember it took a numeric argument and returned a numeric value. Here are some others:

The **ASC( )** function takes a string argument and returns the numeric value of the first character. The numeric value is the decimal representation of the character's ASCII value – an agreed upon world-wide standard. (see http://www.asciitable.com) . Let's try it:

```
PRINT ASC("A")
65
Ready
PRINT ASC("0")
48
Ready
```

The **ABS( )** function takes a number argument and returns it's ABSolute value – if the numeric argument is positive or zero the same number is returned. If the numeric argument is negative the positive value of the number is returned. Let's try it:

```
PRINT ABS(2)
2
Ready
PRINT ABS(0)
0
Ready
PRINT ABS(-3)
3
Ready
```

The **COS( )** function takes a number argument and returns it's COSine value interpreting the number as an angle in degrees. Since the Color LCD ACS Basic only supports whole, integer numbers the number that is returned is equal to the COSine of the angle times 1024. Let's try it. Remember that the COSine of 0 degrees should be 1.0, the COSine of 45 degrees should be 0.707, and the COSine of 90 degrees should be 0.0 – the results are multiplied by 1024 to allow integer manipulation:

```
PRINT COS(0)
1024
Ready
PRINT COS(45)
724
Ready
PRINT COS(90)
0
Ready
```

The **ERR( )** function takes no argument and returns the number of the last error that Basic encountered. This will be useful later on when you want your programs to be able to handle some errors without stopping. There is a table of error number in the ACS Basic Reference at the end of this manual. Let's try it:

```
PRINT ERR()
0
Ready
PRINT 5 / 0
Divide by zero error
Ready
PRINT ERR()
6
Ready
```

The **FIND( )** function takes two arguments – both strings. It returns the zero-based index of the second string in the first, or -1 if the second string doesn't appear in the first string at all. This will be very useful later when you're trying to manipulate strings in your programs. Let's try it:

```
PRINT FIND("Now is the time", "is")
4
Ready
PRINT FIND("Now is the time", "Now")
0
Ready
PRINT FIND("Now is the time", "country")
-1
Ready
```

The **GETCH( )** function takes a number argument telling it how to behave, then returns a number based upon the next character that is entered via the attached communications device. If the argument is 0 the function returns 0 if no character is currently available, otherwise it returns the ASCII decimal value of the character. If the argument is non-zero the function waits for the next character to be received when it then returns the ASCII decimal value of the character.

The **HEX.VAL( )** function takes a string argument which it tries to interpret as a hexadecimal value (only characters 0-9 and A-F, a-f). If it successfully converts the hexadecimal string representation of a number to a value it returns that value – otherwise it causes a Syntax error. A few examples:

```
PRINT HEX.VAL("09AB")
2475
Ready
PRINT HEX.VAL("X91")
Syntax error - can't parse HEX.VAL(argument) to number
Ready
```

The **LEN( )** function takes a string argument and returns its length – how many characters does it contain. This will be very useful later when you're trying to manipulate strings in your programs. Let's try it:

```
PRINT LEN("Now is the time")
15
Ready
PRINT LEN("")
0
Ready
```

The **MULDIV( )** function takes three numeric arguments and returns the result of multiplying the first two together then dividing by the third. What's the value of that? Why can't you just do A * B / C ? The magic is that the three numeric arguments are converted from 32-bit signed integers to 64-bit signed integers first, then the multiply and divide are performed, then the result is converted back to a 32-bit signed integer and returned. Without this function, the only numbers that you could multiply together with a correct result would have to have a product of no more than +2,147,483,647.

The **MULMOD( )** function takes three numeric arguments and returns the result of multiplying the first two together and then taking the modulo with the third. Here's an example of using both MULDIV( ) and MULMOD( ) to calculate 55 percent of 999. Using 16-bit integer math would cause an overflow:

```
list
10 REM calculate 55 percent of 999 (999 * 55) / 100 = 549.45
20 PRINT MULDIV(999,55,100);".";MULMOD(999,55,100)
Ready
run
549.45
Ready
```

The **RGB( )** function takes in three numeric arguments representing three color components – Red, Green and Blue. These are then converted into a 16-bit pixel color that is used for DRAWing on the display.

The **RND( )** function takes in a numeric argument and returns a random number that ranges from zero to the argument minus one.

The **SIN( )** function takes a number argument and returns it's SINe value interpreting the number as an angle in degrees. Since the Color LCD ACS Basic only supports whole, integer numbers the number that is returned is equal to the SINe of the angle times 1024. Let's try it. Remember that the SINe of 0 degrees should be 0.0, the SINe of 45 degrees should be 0.707, and the SINe of 90 degrees should be 1.0 – the results are multiplied by 1024 to allow integer manipulation:

```
PRINT SIN(0)
0
Ready
PRINT SIN(45)
724
Ready
PRINT SIN(90)
1024
Ready
```

The **VAL( )** function takes a string argument which it tries to interpret as a decimal value. If it successfully converts the decimal string representation of a number to a value it returns that value – otherwise it causes a Syntax error.

```
PRINT VAL("12345")
12345
Ready
PRINT VAL("-35")
-35
Ready
```

Functions Returning a String

These functions take one or more arguments and return a string value. You can tell that they return a string value because their function names end with a dollar sign.

The **CHR$( )** function takes a number argument and returns a single character string where the character is the ASCII character of the decimal number. (see http://www.asciitable.com). Let's try it:

```
PRINT CHR$(65)
A
Ready
PRINT CHR$(48)
0
Ready
```

The **ERR$( )** function takes no argument and returns the string of the last error message that Basic encountered. This will be useful later on when you want your programs to be able to handle some errors without stopping. There is a table of error messages in the ACS Basic Reference at the end of this manual. Let's try it:

```
PRINT ERR$()
0 error
Ready
PRINT 5 / 0
Divide by zero error
Ready
PRINT ERR$()
Divide by zero error
Ready
```

The **FMT$( )** function takes two arguments; the first is a string containing information about how to format the second argument into the string value that it returns. The description of the format string argument can be found in the ACS Basic Reference section of this manual. Let's try it:

The **HEX.STR$( )** function takes a numeric argument and returns the hexadecimal string equivalent. Let's try it:

```
PRINT HEX.STR$(25)
19
Ready
PRINT HEX.STR$(256)
100
Ready
```

The **INSERT$( )** function takes three arguments – a source string to insert into, a zero-based offset of where to insert and a string to add into the source string. It returns the source string with the string to be added inserted at the numeric offset. The source string is not directly modified by the function. Let's try it:

```
10 REM test insert$
20 baseString$ ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 insertString$ ="insert"
35 REM insert at beginning
40 PRINT INSERT$(baseString$,0,insertString$)
45 REM insert in middle
50 PRINT INSERT$(baseString$,13,insertString$)
55 REM insert past end
60 PRINT INSERT$(baseString$,30,insertString$)
Ready
run
insertABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMinsertNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZinsert
Ready
```

The **LEFT$( )** function takes two arguments – a source string and the number of characters from the beginning of the string to return. The source string is not directly modified by the function. Let's try it:

```
10 REM test left$
20 baseString$ ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
35 REM take leftmost 5 characters
40 PRINT LEFT$(baseString$,5)
45 REM take leftmost 10 characters
50 PRINT LEFT$(baseString$,10)
Ready
run
ABCDE
ABCDEFGHIJ
Ready
```

The **MID$( )** function takes three arguments – a source string, a zero-based offset of where to start taking from and the number of characters to return. The source string is not directly modified by the function.

The **RIGHT$( )** function takes two arguments – a source string and the number of characters from the end of the string to return. The source string is not directly modified by the function.

The **REPLACE$( )** function takes three arguments.

The **STR$( )** function takes a single numeric argument and returns a string representation of that number.

Some Funny Characters

Your communications device – whether it is a PC or PS/2 keyboard or even the pop-up touch keypad has a limited set of characters. Your program may require some special characters or symbols be displayed – like a degree symbol after a temperature for example. The Color LCD can display these characters that don't appear on the keyboard, but it takes a couple tricks.

The first trick is the understanding that each displayed character is actually based upon a number. Back in the 60's a standards committee got together and hashed out a definition of what number represents what character. It started with the old telegraphic codes for characters and evolved to support the newly designed teleprinters. In computer terminology this is called ***A.S.C.I.I. – which is the abbreviation for American Standard Code for Information Interchange***. It defines 128 characters and control codes, each of which can be uniquely represented by a seven bit number.

There's a chart of these numbers and their representative characters at http://www.asciitable.com . Now when we want to show a certain character we just have to the single 7-bit number to be sent to the display. This means we need a function that takes in a decimal number and converts it to a single number representing the string character based upon this ASCII standard. ACS Basic has just such a function built-in.

The **CHR$( )** function takes a decimal number argument and returns a single character string where the character is the ASCII character of the decimal number. This character can then be sent to the display by PRINTing it. Let's try it:

```
PRINT CHR$(65)
A
Ready
PRINT CHR$(48)
0
Ready
```

So how do we display characters that aren't in the ASCII standard? It turns out that there is another standard call Unicode which defines how to combine multiple non-ASCII codes into a single extended character. Remember that ASCII only defines 7-bit codes, 0 – 127. Since computers work with 8-bit codes, 0 – 255, that leaves codes 128 – 255 available. However the Unicode designers didn't want to limit themselves to just 128 more characters – multiple languages required many more. So they came up with an expandable scheme using multiple prefix characters that can be combined and decoded to represent many more extended characters.

Sounds complicated – and it can be – if you have to do the encoding and decoding. However the Color LCD only provides display support for what is known as the DOS – United States character set. There is a table of these supported characters at the end of this manual. To display them you just have to send the multiple character sequence underneath the character. The definition of what values represent what extended characters were also defined starting in the 60's and is loosely known as the ANSI code standard. ANSI stands for American National Standards Institute.

So let's try showing that degree character. From the table, it looks like the two codes are 194 and 176 – remember to separate the two function calls with a semicolon – if you use a comma instead it won't work because a space character (32) will get inserted between them:

```
print chr$(194);chr$(176)
°
Ready
```

And there it is. You can try some others from the table – some take two codes and some three. Using Basic PRINT commands to display characters is referred to as ANSI operation.

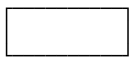
Don't Get Boxed In

If you look at the Extended Characters table at the end of the manual, there are some that can be used for line drawing. Let's try and draw a simple box. We need the four corners, the lines across the top and bottom and the lines on the side. Try typing this short program in – don't forget to NEW first:

```
10000 REM Draw a box using ANSI characters
10005 PRINT CHR$(226); CHR$(148); CHR$(140);
10010 FOR I=1 TO 10
10015 PRINT CHR$(226); CHR$(148); CHR$(128);
10020 NEXT I
10025 PRINT CHR$(226); CHR$(148); CHR$(144)
10030 PRINT CHR$(226); CHR$(148); CHR$(130);
10035 FOR I = 1 TO 10
10040 PRINT " ";
10045 NEXT I
10050 PRINT CHR$(226); CHR$(148); CHR$(130)
10055 PRINT CHR$(226); CHR$(148); CHR$(148);
10060 FOR I=1 TO 10
10065 PRINT CHR$(226); CHR$(148); CHR$(128);
10070 NEXT I
10075 PRINT CHR$(226); CHR$(148); CHR$(152)
```

And now RUN it:

```
run
```



```
Ready

save ansibox
Ready
```

If it works as shown then you can save it for later.

Now you're probably asking yourself why are the line numbers so large? The large line numbers allow us to come back later and put some lower-numbered lines before these. In this example they're going to let us save some typing.

Character Rules

1. Characters are actually a displayed version of a single number.
2. The first 128 characters (numbers 0 – 127) have been agreed upon and are referred to as the ASCII character set. Refer to http://www.asciitable.com
3. There is a built-in function, CHR$( ) that can convert a numeric argument into a character.
4. Additional extended characters may be displayed using multiple character prefix codes.
5. Using Basic PRINT commands on the display is known as ANSI operation.

Don’t Repeat Yourself

Suppose that you want to draw two boxes, one above the other. You could type the same thing in again, duplicating all of these lines. Your programs would tend to get quite large – and, you might run out of line numbers. What you have now is a box-drawing routine. Suppose you could just call this routine whenever you wanted to draw a box.

Good News! You can. There is a routine calling command in most computer languages – including ACS Basic. In computer terminology these common routines are referred to as ***subroutines***. So how do you get to this part of your program from somewhere else? And how do you get back? The new command pair is GOSUB / RETURN. And they are a pair – you can’t have one without the other. The computer terminology is ***calling into a subroutine*** and ***returning from a subroutine*** – call in and return back right after where you called from.

Let’s add some additional lines to the program:

```
10 print “Here’s a box:”

20 gosub 10000

30 print “And here’s another:”

40 gosub 10000

50 end
```

And don’t forget the last line that will turn the box drawing lines into a subroutine:

```
10080 return
```

Your program should now look like this:

```
list
10 PRINT "Here's a box:"
20 GOSUB 10000
30 PRINT "and here's another:"
40 GOSUB 10000
50 END
10000 REM Draw a box using ANSI characters
10005 PRINT CHR$(226); CHR$(148); CHR$(140);
10010 FOR I=1 TO 10
10015 PRINT CHR$(226); CHR$(148); CHR$(128);
10020 NEXT I
10025 PRINT CHR$(226); CHR$(148); CHR$(144)
10030 PRINT CHR$(226); CHR$(148); CHR$(130);
10035 FOR I = 1 TO 10
10040 PRINT " ";
10045 NEXT I
10050 PRINT CHR$(226); CHR$(148); CHR$(130)
10055 PRINT CHR$(226); CHR$(148); CHR$(148);
10060 FOR I=1 TO 10
10065 PRINT CHR$(226); CHR$(148); CHR$(128);
10070 NEXT I
10075 PRINT CHR$(226); CHR$(148); CHR$(152)
10080 RETURN
Ready
```

And when you run it:

```
run
Here's a box:
┌──────────┐
│          │
└──────────┘
and here's another:
┌──────────┐
│          │
└──────────┘
Ready
```

If you change the GOSUB to a GOTO you will get an error when your program executes the RETURN command – it doesn't know the way back that is remembered by the GOSUB.

Subroutines are used to replace common or duplicate code. If you look at our program there's another common piece of code that could be made into a subroutine – the PRINTing of the Unicode prefix for the box drawing extended characters: PRINT CHR$(225); CHR$(148);

Let's make this repetitive code into a separate subroutine – it could go above or below, we'll put it below:

```
10100 REM Print the Unicode prefix characters
10105 PRINT CHR$(226); CHR$(148);
10110 RETURN
```

```
list
10 PRINT "Here's a box:"
20 GOSUB 10000
30 PRINT "and here's another:"
40 GOSUB 10000
50 END
10000 REM Draw a box using ANSI characters
10005 PRINT CHR$(226); CHR$(148); CHR$(140);
10010 FOR I=1 TO 10
10015 PRINT CHR$(226); CHR$(148); CHR$(128);
10020 NEXT I
10025 PRINT CHR$(226); CHR$(148); CHR$(144)
10030 PRINT CHR$(226); CHR$(148); CHR$(130);
10035 FOR I = 1 TO 10
10040 PRINT " ";
10045 NEXT I
10050 PRINT CHR$(226); CHR$(148); CHR$(130)
10055 PRINT CHR$(226); CHR$(148); CHR$(148);
10060 FOR I=1 TO 10
10065 PRINT CHR$(226); CHR$(148); CHR$(128);
10070 NEXT I
10075 PRINT CHR$(226); CHR$(148); CHR$(152)
10080 RETURN
10100 REM Print the Unicode prefix characters
10105 PRINT CHR$(226); CHR$(148);
10110 RETURN
Ready
```

And now we have to change lines 10005, 10015, 10025, 10030, 10050 10055, 10065 and 10075. You could type these lines over, but as we'll see soon, ACS Basic has a shortcut – the EDIT command.

Rules for Subroutines

1. Any block of code can be turned into a subroutine by adding a RETURN command as the last line and then calling the code using the GOSUB command.
2. GOSUB commands must be paired with RETURN commands – you can't GOTO in or out of a subroutine. While it may appear to work, the stack that Basic uses to remember where to RETURN to will eventually become confused and an error will occur. To leave a subroutine, GOTO the subroutine's RETURN statement.
3. Subroutines can be nested – code inside a subroutine can call another subroutine.

Making Changes

Type:

```
EDIT 10005
```

Notice how Basic prints out the line, but stays at the end – it doesn't show the Ready prompt. The cursor is blinking at the end of the line. It's waiting for you to modify the line and tell it when you're done.

```
edit 10005
10005 PRINT CHR$(226); CHR$(148); CHR$(140); _
```

You can move your blinking cursor left and right with the keyboard left and right arrow keys ←. Some communication devices will also allow you to Home home to the beginning of the line or End end to the end of the line. You can delete characters left of the cursor using the backspace key backspace. If you want to quit editing without saving your changes use the double ESCape esc esc key. When you want to stop editing and save your changes press the enter key Enter←.

One thing to remember about EDITing – you're always in 'insert' mode. Any printable character that you type will be inserted into the line wherever the cursor is blinking. If you make a mistake, you can use the backspace key and retype it, or use the arrow keys to move after it, backspace and retype.

Use the left arrow key to move the cursor from the end of the line to the C in the last CHR$(140) function. Now press the backspace key to remove characters until the space after the PRINT. Again use the left arrow key to move the cursor to the P in the PRINT. The type in GOSUB 10100: - the colon is important, it allows you to group multiple commands on the same line. When you're done making these changes the line should look like:

```
10005 GOSUB 10100 : PRINT CHR$(140);
```

If it doesn't look like this use the arrow keys, backspace and typing to correct it. When it does look like this press Enter. The line should be accepted as if you had typed it in again instead of editing. You can look at it using the LIST command with the line number:

```
list 10005
10005 GOSUB 10100 : PRINT CHR$(140);
Ready
```

Now try running the program. It should work the same as it did before the change.

Edit the other lines to make the same change. Once you get the hang of using the EDIT command you'll probably find it much easier than typing the complete line over again, and you'll tend to make fewer mistakes.

```
list
10 PRINT "Here's a box:"
20 GOSUB 10000
30 PRINT "and here's another:"
40 GOSUB 10000
50 END
10000 REM Draw a box using ANSI characters
10005 GOSUB 10100 : PRINT CHR$(140);
10010 FOR I=1 TO 10
10015 GOSUB 10100 : PRINT CHR$(128);
10020 NEXT I
10025 GOSUB 10100 : PRINT CHR$(144)
10030 GOSUB 10100 : PRINT CHR$(130);
10035 FOR I = 1 TO 10
10040 PRINT " ";
10045 NEXT I
10050 GOSUB 10100 : PRINT CHR$(130)
10055 GOSUB 10100 : PRINT CHR$(148);
10060 FOR I=1 TO 10
10065 GOSUB 10100 : PRINT CHR$(128);
10070 NEXT I
10075 GOSUB 10100 : PRINT CHR$(152)
10080 RETURN
```

```
10100 REM Print the Unicode prefix characters
10105 PRINT CHR$(226); CHR$(148);
10110 RETURN
Ready
run
Here's a box:
┌──────────┐
│          │
└──────────┘
and here's another:
┌──────────┐
│          │
└──────────┘
Ready
```

And finally, now that you know you can have multiple statements on a line, you can shorten this up some more while still keeping it readable:

```
list
10 PRINT "Here's a box:" : GOSUB 10000
30 PRINT "and here's another:" : GOSUB 10000
50 END
10000 REM Draw a box using ANSI characters
10005 GOSUB 10100 : PRINT CHR$(140);
10010 FOR I = 1 TO 10 : GOSUB 10100 : PRINT CHR$(128); : NEXT I
10025 GOSUB 10100 : PRINT CHR$(144)
10030 GOSUB 10100 : PRINT CHR$(130);
10035 FOR I = 1 TO 10 : PRINT " "; : NEXT I
10050 GOSUB 10100 : PRINT CHR$(130)
10055 GOSUB 10100 : PRINT CHR$(148);
10060 FOR I=1 TO 10 : GOSUB 10100 : PRINT CHR$(128); : NEXT I
10075 GOSUB 10100 : PRINT CHR$(152)
10080 RETURN
10100 REM Print the Unicode prefix characters
10105 PRINT CHR$(226); CHR$(148); : RETURN
Ready
```

Rules for Editing

1. You can modify program lines in-place using the EDIT command.
2. While EDITing a program line, the arrow keys move the blinking cursor left and right and the backspace key deletes the character to the left.
3. While EDITing a program line any other character that you type will be entered into the line where the blinking cursor is – you are always in 'insert mode'.
4. You can discard your changes while editing by ESCaping – pressing the ESC key twice in a row.
5. You can accept your editing changes by pressing Enter – you don't have to be at the end of the line.
6. Changing the line number of an EDITed line will add or replace the newly numbered line in your program – the original line will remain in-place and intact.

There's a System

So far, we've only defined our own variables – numeric and string, simply by using them in our programs to remember values. We've used commands and operators to access, modify and compare their contents. And we've used built-in functions like read-only variables to extend our program operation with built-in functionality. What's next?

Just like the built-in functions, ACS Basic has some built-in variables to access and modify values used by the Color LCD 'system'. These are called System Variables and they use a special naming so that they don't interfere with your program variables. So what can these System Variables do for you?

How about timing? Let's try to make a simple metronome application. Type NEW and enter the following small program:

```
10 REM Metronome
15 t = 0 : INPUT "Enter tempo (1 - 32766) :",tempo
20 IF tempo < 1 OR tempo > 32766 THEN GOTO 15
25 FOR i = 1 TO tempo : d = d + 1 : NEXT i
30 IF t = 0 THEN PRINT "Tick" ELSE PRINT "Tock"
35 IF t = 0 THEN t = 1 ELSE t = 0
40 GOTO 25
Ready
```

Let's take a moment and see what this program is doing. Line 10 is a REMark – a comment – telling us what this program is.

Line 15 initializes the tick/tock variable – this isn't really necessary since variables are set to 0 the first time they're used, but it's good practice to start with a known value. The program user is then prompted to enter a number for the tempo – notice that the INPUT command can have an optional prompt string that it will display before waiting for the user to enter a value and press enter.

Line 20 checks the tempo that the user entered and, if it's not within range, requests that the user enter a different value by looping back to line 15. It's always a good idea to validate user input to your programs. Notice that it performs two comparisons on the tempo value and then combines them logically with the OR logical operator – IF the first condition OR the second condition is True THEN loop back to line 15.

Lin 25 counts from one to the tempo value – it does nothing but waste time. The amount of time that must pass before the FOR / NEXT counting loop ends is dependent upon the tempo value entered. The d=d+1 statement is there solely to slow the loop down so reasonable tempo values have an effect.

Line 30 prints Tick or Tock depending upon the tick/tock variable t. Line 35 toggles the tick/tock variable t between zero and one.

Line 40 loops back to the delay loop.

Try running the program and entering different values for the tempo – a value of 4000 seems to be close to 60 beats per minute – once a second. You will have to ESCape the program to try another tempo:

```
run
Enter tempo (1 - 32766) :4000
Tick
Tock
Tick
Tock
Tick
Tock
Tick
Tock
Tick
 ESC at line 25
Ready
```

So how can we improve this program? It sure would be more useful to just enter the beats per minute instead of an arbitrary number. And this program kind of works backwards – a higher tempo value should result in a faster beats per minute – not slower.

We could try different equations to compute a delay loop count that corresponds to the requested tempo value but that would take a lot of trial and error to get the correct calculation that would relate the two values. A better idea would be to use some kind of timer that we could control.

System Timers

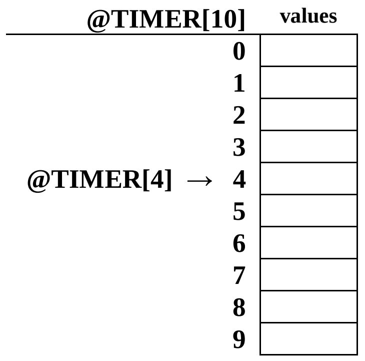
The Color LCD system has such a timer – in fact, there are ten of them. How do you access them and how do they work?

In ACS Basic, a system variable is identified by preceding the name with the '@' character. To access the first timer you would write:

```
@TIMER[0] = value : REM sets timer 0

value = @TIMER[0] : REM gets timer 0
```

There is an array of ten timers: numbered 0 – 9. You might have guessed that the zero in brackets is telling Basic that you want timer number 0. The brackets [ ] indicate that you are selecting a single timer – the one identified by the number in-between. In computer terminology this is referred to as ***indexing into an array***.



So how do these timers work? Fifty times per second, the Color LCD checks these ten timers. If any of them are not equal to zero then the display subtracts one from it. In computer terminology this is referred to as ***decrementing a non-zero timer***. So you start the timer by setting it to a non-zero value, then you can 'monitor' the timer by checking to see if it is at zero. Since it counts down by 50 times per second, to set the timer for one second you would set it to fifty. Cool.

So how do we set a timer count for beats per minute? We can calculate this given the counts per second and the seconds per minute to get the counts per beat:

$$\frac{\left(50\,\frac{counts}{second} \times 60\,\frac{seconds}{minute}\right)}{\frac{beats}{minute}} = 3000\,\frac{counts}{beats}$$

OK. Divide 3000 by the desired beats per minute and use that to set the timer. Then we can loop for the timer to be zero for our delay.

As for validating the user input, we can't divide by zero – it's an error so we have to be greater than that, and dividing 3000 by anything larger than 3000 will result in a zero – timer won't run at all. Reasonable numbers are probably greater than 30 bpm and less than 300 bpm. Let's change the program:

```
10 REM Metronome
15 t = 0 : INPUT "Enter tempo beats per minute (30 - 300) :",tempo
20 IF tempo < 30 OR tempo > 300 THEN GOTO 15
25 @TIMER[0] = 3000 / tempo
27 IF @TIMER[0] > 0 THEN GOTO 27
30 IF t = 0 THEN PRINT "Tick" ELSE PRINT "Tock"
35 IF t = 0 THEN t = 1 ELSE t = 0
40 GOTO 25
```

And you can run it and try different tempo values.

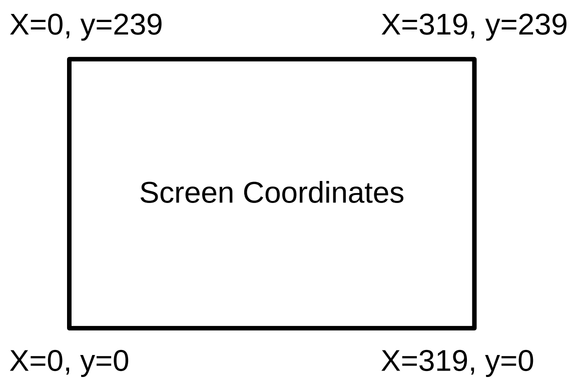
Rules for System Variables

1. System variables access and control Color LCD system functions.
2. System variable names are prefixed with an '@' character.
3. System variables can have different meanings if they are read versus written.
4. Some system variables are read-only.
5. Some system variables are implemented as an array and require an index for selection.

Enough Text – Let's Draw Something

Up to now we have been PRINTing in ANSI operation. That's nice, but what about some graphics. The Color LCD and ACS Basic provide several commands to draw graphics on the screen.

The graphic drawing commands use pairs of numbers to identify where you want to draw on the display. In computer terminology these are referred to as ***screen coordinates***. The first number of the pair is the X coordinate and identifies a horizontal starting point that ranges from zero to one less than the width of the display (0 – 319). The second number of the pair is the Y coordinate and identifies a vertical starting point that ranges from zero to one less than the height of the display (0 – 239). The X and Y coordinates for drawing start at **0, 0** at the lower left corner of the display and range to 319, 239 at the upper right corner of the display as shown in this diagram:



Drawing is clipped at the coordinate boundaries of the display – commands that specify coordinates outside of these boundaries will not affect the display when either coordinate exceeds these values.

Pixels

The screen is divided up into thousands of dots arranged in columns (0 – 319) and rows (0 – 239) that can be individually colored with one of 65536 colors. In computer terminology these dots are referred to as ***pixels***. The pixel color is selected by specifying the amount of three pure color components that are added to form the desired color – the amount of Red, the amount of Green and the amount of Blue. In computer terminology this is referred to as an ***additive RGB color model***.

The Color LCD supports 32 Red levels (0 – 31), 64 Green levels (0 – 63) and 32 Blue levels (0 – 31) for each pixel. There are more green levels than red or blue because the human eye is most sensitive to changes in green. Thirty two unique values can be determined using 5 binary bits, and sixty three unique values can be specified using 6 binary bits – so this form of RGB color encoding is known as RGB565. Hey – the total number of bits is 16 – the same number in our integer arithmetic! So, since the colors are additive, black must be equal to zero, and white must be equal to 65535 (largest 16-bit number).

The ACS Basic drawing commands allow your program to control the colors of one or more pixels at a time, depending upon the command, to draw graphic objects. The command to draw a single pixel is the simplest command – this table shows how the command is constructed and what the command arguments are (don't type this in directly, it's a description showing you how to construct this command):

DRAW.PIXEL x,y,color		
Sets the pixel at x,y on the current drawing @SURFACE to color	x	screen x coordinate (0 - 319)
	y	screen y coordinate (0 - 240)
	color	RGB565 pixel color

Let's use this description to construct a command that draws a white pixel at the center of the screen – don't worry about the @SURFACE system variable for now – we need to specify the x and y coordinates and the desired pixel color:

```
draw.pixel 160,120,65535
```

The white pixel is probably hard to see amongst the text on the screen – you could draw a red pixel that might be easier to see but how to you specify the red color? There's a built-in function for that.

The RGB(r, g, b) function takes three arguments, red level r (0 – 255), green level g (0 – 255) and blue level b (0 – 255) and does the math to combine them into a 16-bit integer – which is its return value. So let's use this to draw a red pixel:

```
draw.pixel 160,120,rgb(255,0,0)
```

Kind of hard to see – the individual pixels are kind of small and there's a lot of clutter on the screen. Let's use another system variable to stop showing the program text and PRINT output on the small screen – they will still show on your connected communications device. The ANSI operating mode can be turned off by setting the @ANSI.ENABLE system variable to zero:

```
@ANSI.ENABLE=0
```

Now as you type, or PRINT, the output doesn't affect the Color LCD screen. As a default, so nobody wonders where their PRINT output, error message or Ready prompt went, the @ANSI.ENABLE is set to one whenever a program is run or stops running. So if you don't want ANSI output on the Color LCD screen while your program is running you will have to turn it off at the beginning of your program, and keep your program running.

Clearing The Screen

One last thing – how do we clear the screen so we can see the pixel? Turns out that there's a command for that to – in computer terminology this is referred to a ***filling the screen***. Here's the command format:

DRAW.FILL x, y, width, height, color		
Sets the width by height pixels starting at x, y on the current drawing @SURFACE to color	x	screen x coordinate (0 - 319)
	y	screen y coordinate (0 - 240)
	width	number of pixels wide to fill
	height	number of pixels high to fill
	color	RGB565 fill pixels color

So try it:

```
draw.fill 0, 0, 320, 240, RGB(0, 0, 0)
```

You can experiment with different positions, widths, heights and colors. So now we have enough for a simple graphics program:

```
10 REM Draw a red pixel at the center of the screen
15 @ANSI.ENABLE=0
20 DRAW.FILL 0,0,320,240,RGB(0,0,0)
25 DRAW.PIXEL 160,120,RGB(255,0,0)
30 GOTO 30
```

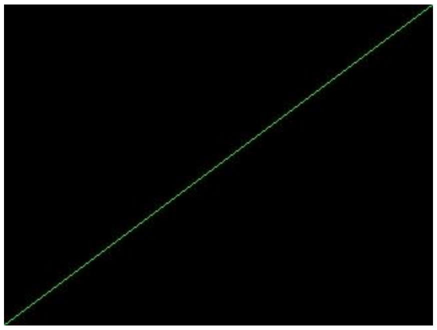
Drawing Lines

How about drawing lines? Let's try drawing a green line diagonally from one corner of the screen to the other. Here's the command format:

DRAW.LINE sx, sy, ex, ey, color		
Draws a line from sx, sy to ex, ey on the current drawing @SURFACE using color	sx	starting x coordinate (0 – 319)
	sy	starting y coordinate (0 – 240)
	ex	ending x coordinate (0 – 319)
	ey	ending y coordinate (0 – 240)
	color	RGB565 line pixels color

Here's the program:

```
10 REM Draw a green line from lower-left to upper-right
15 @ANSI.ENABLE=0
20 DRAW.FILL 0,0,320,240,RGB(0,0,0)
25 DRAW.LINE 0,0,319,239,RGB(0,255,0)
30 GOTO 30
```



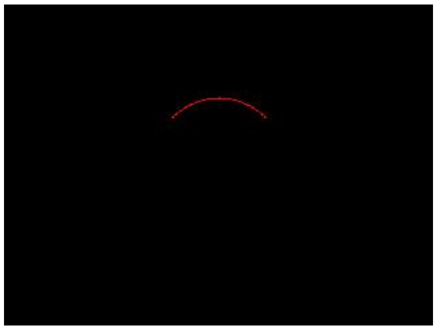
Drawing Arcs

How about drawing arcs? You could compute and draw each pixel making up the arc, or have ACS Basic draw it for you if you provide the center, width, height and starting/ending angles. Here's the command format:

DRAW.ARC x,y,width,height,start,end,color		
Draws an arc centered at x,y of width,height beginning at start angle and ending on end angle on the current drawing @SURFACE using color	x	center x coordinate (0 – 319)
	y	center y coordinate (0 – 240)
	width	arc width (0 – 319)
	height	arc height (0 – 240)
	start	starting angle (0 - 359)
	end	ending angle (0 – 359)
	color	RGB565 box pixels color

Here's the program:

```
10 REM draw a red 90 degree arc at the screen center
15 @ANSI.ENABLE=0
20 DRAW.FILL 0,0,320,240,RGB(0,0,0)
25 DRAW.ARC 160,120,100,100,45,135,RGB(255,0,0)
30 GOTO 30
```



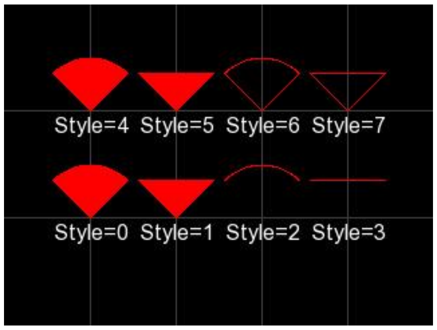
Drawing Arcs with Style

There's another arc drawing command that allows the arc to be 'styled'. Here's the command format:

DRAW.ARC.STYLED x,y,width,height,start,end,color,style		
Draws a 'styled' arc centered at x,y of width,height beginning at start angle and ending on end angle on the current drawing @SURFACE using color	x	center x coordinate (0 – 319)
	y	center y coordinate (0 – 240)
	width	arc width (0 – 319)
	height	arc height (0 – 240)
	start	starting angle (0 - 359)
	end	ending angle (0 – 359)
	color	RGB565 box pixels color
	style	combination of zero or more style bits that are 'or'ed together (0 – 7): 1 – Chord 2 – No Fill 4 - Edged

Here's a program to show the effect of the various combinations of the style bits:

```
10 REM arc demo
20 @ANSI.ENABLE=0
30 REM draw grid
40 FOR x = 64 TO 256 STEP 64 : DRAW.LINE x,0,x,239,RGB(64,64,64) : NEXT x
50 FOR y = 80 TO 160 STEP 80 : DRAW.LINE 0,y,319,y,RGB(64,64,64) : NEXT y
60 REM init text font
70 @FONT.HALIGN[0]=1 : @FONT.VALIGN[0]=2 : @FONT.FCOLOR[0]=RGB(255,255,255)
80 REM draw eight styles of filled arcs
90 DRAW.ARC.FILLED 64,80,80,80,45,135,RGB(255,0,0),0
100 DRAW.TEXT 0,64,80,"Style=0"
110 DRAW.ARC.FILLED 128,80,80,80,45,135,RGB(255,0,0),1
120 DRAW.TEXT 0,128,80,"Style=1"
130 DRAW.ARC.FILLED 192,80,80,80,45,135,RGB(255,0,0),2
140 DRAW.TEXT 0,192,80,"Style=2"
150 DRAW.ARC.FILLED 256,80,80,80,45,135,RGB(255,0,0),3
160 DRAW.TEXT 0,256,80,"Style=3"
170 DRAW.ARC.FILLED 64,160,80,80,45,135,RGB(255,0,0),4
180 DRAW.TEXT 0,64,160,"Style=4"
190 DRAW.ARC.FILLED 128,160,80,80,45,135,RGB(255,0,0),5
200 DRAW.TEXT 0,128,160,"Style=5"
210 DRAW.ARC.FILLED 192,160,80,80,45,135,RGB(255,0,0),6
220 DRAW.TEXT 0,192,160,"Style=6"
230 DRAW.ARC.FILLED 256,160,80,80,45,135,RGB(255,0,0),7
240 DRAW.TEXT 0,256,160,"Style=7"
999 GOTO 999
```


Style=4 Style=5 Style=6 Style=7
Style=0 Style=1 Style=2 Style=3

Drawing Boxes

How about drawing boxes? You could compute and draw each line making up the box, or have ACS Basic draw it for you if you provide the locations of two diagonal corners. Here's the command format:

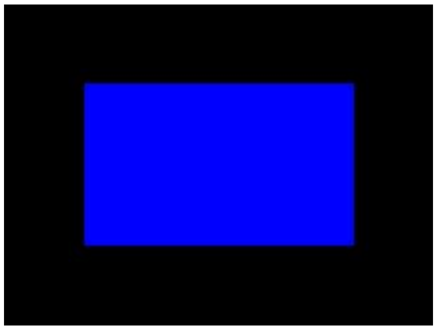
DRAW.BOX x1,y1,x2,y2,color		
Draws a box from diagonal corners x1,y1 to x2,y2 on the current drawing @SURFACE using color	x1	corner 1 x coordinate (0 – 319)
	y1	corner 1 y coordinate (0 – 240)
	x2	corner 2 x coordinate (0 – 319)
	y2	corner 2 y coordinate (0 – 240)
	color	RGB565 box pixels color

Here's the program:

```
10 REM Draw a blue box around the screen edge
15 @ANSI.ENABLE=0
20 DRAW.FILL 0,0,320,240,RGB(0,0,0)
25 DRAW.BOX 0,0,319,239,RGB(0,0,255)
30 GOTO 30
```

and if you want a filled box:

```
10 REM Draw a blue box around the screen edge
15 @ANSI.ENABLE=0
20 DRAW.FILL 0,0,320,240,RGB(0,0,0)
25 DRAW.BOX.FILLED 60,60,260,180,RGB(0,0,255)
30 GOTO 30
```



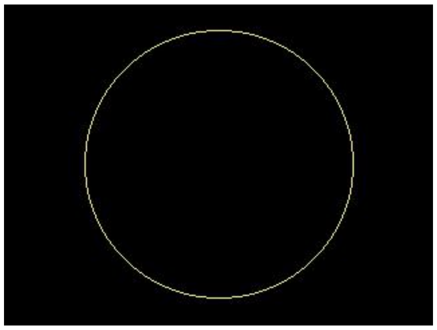
Drawing Circles

How about drawing a circle? Here's the command format:

DRAW.CIRCLE x,y,r,color		
Draws a circle of radius r centered at x, y on the current drawing @SURFACE using color	x	center x coordinate (0 – 319)
	y	center y coordinate (0 – 240)
	r	circle radius(0 – 319)
	color	RGB565 circle pixels color

and here's a program to draw a yellow circle centered on the screen:

```
10 REM Draw a blue box around the screen edge
15 @ANSI.ENABLE=0
20 DRAW.FILL 0,0,320,240,RGB(0,0,0)
25 DRAW.CIRCLE 160,120,100,RGB(255,255,0)
30 GOTO 30
```



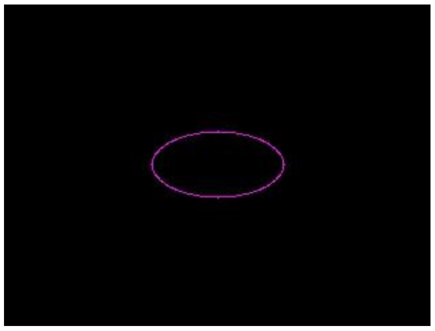
Drawing Ellipses

How about drawing an Ellipse? Here's the command format:

DRAW.ELLIPSE x, y, width, height, color		
Draws a width by height ellipse centered at x, y on the current drawing @SURFACE to color	x	center x coordinate (0 – 319)
	y	center y coordinate (0 – 240)
	width	major axis width
	height	minor axis width
	color	RGB565 ellipse pixels color

and here's a program to draw a magenta ellipse centered on the screen:

```
10 REM Draw a blue box around the screen edge
15 @ANSI.ENABLE=0
20 DRAW.FILL 0,0,320,240,RGB(0,0,0)
25 DRAW.ELLIPSE 160,120,100,50,RGB(255,0,255)
30 GOTO 30
```



If you want a filled ellipse change line 25:

```
25 DRAW.ELLIPSE.FILLED 160,120,100,50,RGB(255,0,255)
```

Drawing Polygons

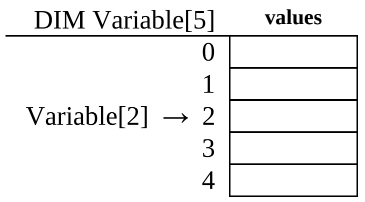
A polygon is a flat shape consisting of straight lines that are joined to form a closed chain. A triangle is just a 3-sided polygon, a box is a 4-sided polygon with right angles between the sides. An N-sided polygon requires N coordinate pairs for its vertices.

How can we format a polygon draw command that can take a variable number of coordinates? The answer is to use an array of numbers to store each coordinate and just provide the array names to the polygon command.

Array variables are just a table of numbers or strings. In order to use an array you have to tell Basic how big of a table that you want. The Color LCD then sets aside the correct amount of memory for that variable to hold the number of values that you require. In computer terminology this is referred to as ***dimensioning an array***. Basic provides a DIMension command for this purpose:

```
30 DIM x[3],y[3]
```

When you access the individual values in this array you specify an index into the table to select which value that you want. The index follows the variable name and is enclosed in brackets [ ]. Here's a conceptual picture of what a 5 element numeric array looks like. Notice that there are 5 elements, numbered 0 – 4:

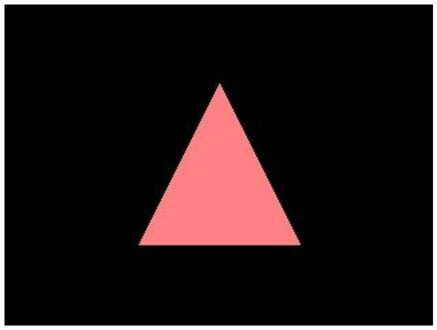


Since each array variable holds a single set of values, to draw a polygon you will need two arrays: one for the X coordinates of each vertex and one for the Y coordinates. Your program 'fills' these arrays with the coordinates for each point – prior to calling the DRAW.POLYGON command. The command looks at the dimensioned size of these coordinate arrays to determine how many sides (and vertices) that the polygon should be drawn with.

So let's draw a 3-sided polygon – a triangle – centered on the screen. We need X and Y coordinate arrays that each hold 3 values. Here's the program:

```
10 REM Draw a triangle
20 @ANSI.ENABLE=0
30 DIM x[3],y[3] : REM Allocate coordinate arrays
40 DRAW.FILL 0,0,320,240,RGB(0,0,0)
50 REM Load x & y coordinates
60 x[0]=100:y[0]=60:x[1]=220:y[1]=60:x[2]=160:y[2]=180
70 DRAW.POLYGON x,y,RGB(255,128,128)
80 GOTO 80
```

And you can also fill it in – change the DRAW.POLYGON command to DRAW.POLYGON.FILLED.



Drawing Text

Let’s say that we need to put text on the screen along with our graphics. You could leave the @ANSI.ENABLE system variable set to one, and use PRINT commands. How do you control where the text appears on the screen?

The @ANSI command has a few additional modifiers – two of which allow you to read the current ANSI row and column values, or set them before printing: @ANSI.ROW and @ANSI.COL. The ANSI rows and columns are numbered like this starting in the upper left of the screen:

	Col 0	Col 1	Col 2	…	Col 42	Col43	Col 44
Row 0							
Row 1				…			
Row 2							
…				…			
Row 17							
Row 18				…			
Row 19							

Try it out – PRINT Hello in the middle of the screen:

```
@ANSI.ROW = 10 : @ANSI.COL = 17 : PRINT “Hello!”
```

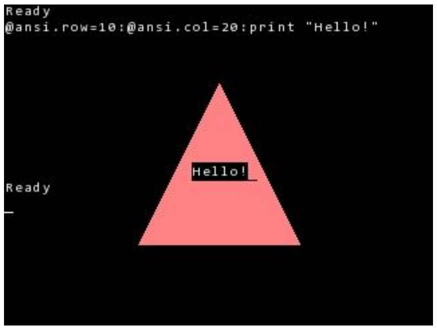
While this does work – there are a few problems. The cursor appears, blinking, on the screen – which you may not want. You can turn this off with another @ANSI command:

```
@ANSI.CURSOR = 0 : REM Turn off ANSI cursor
```

And, if you PRINT on the bottom line of the screen without a trailing semi-colon – or if the text wraps to the next line the screen will scroll up. You can turn these features off with another couple @ANSI commands:

```
@ANSI.WRAP = 0 : REM Turn off ANSI line wrap
@ANSI.SCROLL = 0 : REM Turn off ANSI screen scrolling
```

While this method of drawing text can be made to work, the positioning is coarse, the characters are always white on black, and you are stuck with the character height and style that ANSI operation uses. We’ll examine an alternative text display method next which uses another DRAW command along with some additional system commands to control the text appearance.



```
Ready
@ansi.row=10:@ansi.col=20:print "Hello!"
Hello!
Ready
```

Drawing Text with Style

The shape, appearance and size of drawn text characters specify the text's style. In computer terminology the style of text is referred to as a ***font***. The Color LCD and ACS Basic provide support for up to 32 different fonts to be in use at a time. Each of the 32 fonts can be configured by your program to change the text color, relative position and appearance. The entries in this font table are accessed through system variables. The individual character shapes and their size are obtained from another table in memory by their name – these are the actual embedded fonts. The two pieces of information: the embedded font to use and how it is colored and positioned are selected as an entry in the font table used to draw the desired text. In computer terminology this is referred to as ***rendering text***.

The embedded fonts must be loaded into the display's memory in order to be used for drawing text. They are stored in a collection with other items where they can be quickly accessed by their name.

It's a Matter of Resources

In the Color LCD there is a collection of fonts and other objects that the Basic can draw with – they are collectively known as Resources. The Color LCD looks for resources to load in two places when it starts-up: if there is a micro SD card installed it tries to load a binary file named Resources.bin, and if that fails it tries to load the resources from the on-board flash memory chip.

You can see this process happen when the Color LCD is powered on or reset – the status of the resource search and load is displayed on the top few lines of the screen.

You can list the resources that are currently loaded, the command is RESOURCES.LIST. Here is what the list would look like if no additional resources could be found and loaded at power-up:

```
resources.list
AcsAnsiFont.efnt                      12,604 builtin
AcsDefaultFont.efnt                   27,940 builtin
ButtonK.bmp                            1,610 builtin
ButtonK2.bmp                           2,050 builtin
ButtonK4.bmp                           3,018 builtin
ButtonM.bmp                            2,618 builtin
Ready
```

There are a few resources that are built-in: they are required to allow the status display, Configuration Settings screen and other features to operate if no additional resources can be found. The files with the .EFNT extension are 'embedded font files'. They contain the information about the size and shape of the text characters for drawing purposes. The AcsAnsiFont.efnt file is used for PRINTing ANSI text, the AcsDefaultFont.efnt is used for displaying the power-up status messages and labeling the keys on the Configuration Settings keypad.

There is a Windows utility that can be used to generate these embedded font files from existing computer font files – you select the computer font, what size to make it, what characters to include in the font and what name to give the generated .EFNT file and the utility generates the file. This is described in an Appendix of the Color LCD 320x240 Display Terminal User's Manual.

How do you get your embedded font files into the Color LCD resources? You can then place the embedded font file on the micro SD card and load it into the resources manually or in your program using the RESOURCES.ADD command:

```
resources.add ComicSansBold14.efnt
Ready
resources.list
ComicSansBold14.efnt                  25,666
AcsAnsiFont.efnt                      12,604 builtin
AcsDefaultFont.efnt                   27,940 builtin
ButtonK.bmp                            1,610 builtin
ButtonK2.bmp                           2,050 builtin
ButtonK4.bmp                           3,018 builtin
ButtonM.bmp                            2,618 builtin
Ready
```

So there are at least two embedded font files to work with that are built-in. How can we use these to draw text on the screen? Let's look using the FONTS commands.

There's a Table of Fonts

Just like with the resources, there is a FONTS.LIST command. Try it:

```
fonts.list
F#, "Resource Name                   ", BCOLOR          , FCOLOR          , SPACINGX, SPACINGY, HALIGN, VALIGN, BTRANS, FTRANS
 0, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
 1, "AcsAnsiFont.efnt                ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      0,      1,      1,      0
 2, "AcsAnsiFont.efnt                ", RGB(255,255,255), RGB(  0,  0,  0),        0,        0,      0,      1,      0,      0
 3, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      0,      1,      0
 4, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      0,      0,      1,      0
 5, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
 6, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
 7, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
 8, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
 9, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
10, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
11, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
12, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
13, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
14, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
15, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
16, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
17, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
18, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
19, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
20, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
21, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
22, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
23, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
24, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
25, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
26, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
27, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
28, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
29, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
30, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(255,255,255),        0,        0,      1,      1,      1,      0
31, "AcsDefaultFont.efnt             ", RGB(  0,  0,  0), RGB(  0,  0,  0),        0,        0,      1,      1,      1,      0
Ready
```

What are all of these numbers? The first column is the 'font number' – the index into the font table. In Basic you can treat this table as an array using an index to access individual rows.

The second column is the embedded font file name in the resources that will be used to render text using this font table entry. Note that when setting the .efnt resource to be used by an entry, that entry has to exist in the resources – if it doesn't an error will occur.

The next two columns indicate what Background color (BCOLOR) and Foreground color (FCOLOR) will be used when rendering the text. The FCOLOR is the color of the actual character pixels, the BCOLOR is the color of the pixels surrounding the character. The values in the table are 16-bit RGB565 values – they are shown in the listing as if they were set using the RGB( ) function – which they can be.

The next four columns control the relative positioning of the characters when text is rendered with this font entry. The SPACINGX column controls the spacing between characters of the font. A value of '0' will use the fixed width of the font to place the following character. A '-1' will use the "bounding box" character width for horizontal spacing. A non-zero, positive value will force that spacing character to character when fonts are oriented horizontally.

The SPACINGY column works in the same way but affects vertical spacing when fonts are oriented vertically. (this operation is not currently supported)

The ALIGN columns control the relative placement of the drawn text relative to the starting coordinate provided the DRAW.TEXT command – or the text alignment to the coordinates. The HALIGN entry controls the horizontal alignment: a value of '0' justifies the text from the left, a value of '1' centers the text and a value of '2' justifies the text from the right. The VALIGN entry controls the vertical alignment: a value of '0' justifies the text from the top, a value of '1' centers the text and a value of '2' justifies the text from the bottom.

The last two columns control the transparency of the rendered text. A transparent pixel is not drawn with the result that anything it is drawn on top of will 'show through'. The BTRANS column controls the character background pixels and the FTRANS column controls the foreground pixels. A value of '0' forces those pixels to be drawn, a value of '1' forces those pixels to not be drawn.

All of these font table entries may be accessed – read / written using system variables – specifying an index of the entry:

System Variable	Values	Description
@FONT.EFNT$[entry]	"resource.efnt"	Gets/Sets embedded font resource for entry – the resource must exist in the resource table to set
@FONT.BCOLOR[entry]	RB565 pixel color	Gets/Sets font background color for entry – use RGB( ) to set
@FONT.FCOLOR[entry]	RB565 pixel color	Gets/Sets font foreground color for entry – use RGB( ) to set
@FONT.SPACINGX[entry]	-1 = bounding box 0 = fixed width >0 = spacing	Gets/Sets font horizontal spacing
@FONT.SPACINGY[entry]	-1 = bounding box 0 = fixed width >0 = spacing	Gets/Sets font vertical spacing (not currently supported)
@FONT.HALIGN[entry]	0 = left justify 1 = center justify 2 = right justify	Gets/Sets font horizontal alignment
@FONT.VALIGN[entry]	0 = top justify 1 = center justify 2 = bottom justify	Gets/Sets font vertical alignment
@FONT.BTRANS[entry]	0 = normal 1 = transparent	Gets/Sets font background transparency
@FONT.FTRANS[entry]	0 = normal 1 = transparent	Gets/Sets font foreground transparency

And Now Back to Drawing the Text

So now we can get back to the DRAW.TEXT command. As with the other commands here's the command format:

DRAW.TEXT f, x, y, expr		
Renders the expression on the current drawing @SURFACE relative to x, y using the font table entry f values	f	font table entry number (0 – 31)
	x	justification x coordinate (0 – 319)
	y	justification y coordinate (0 – 240)
	expr	the number or string to draw

So let's try this out. There's already some default entries in the font table. What if we write a simple program that draws "Hello!" in the center of the screen using Font zero:

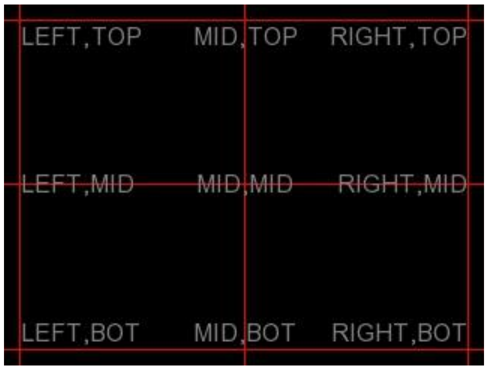
```
10 REM Draw text
20 @ANSI.ENABLE=0
30 DRAW.TEXT 0,160,120,"Hello!"
40 GOTO 40
```

Let's make the font red; set Font zero's foreground color to red before drawing the text:

```
10 REM Draw red text
20 @ANSI.ENABLE=0
25 @FONT.FCOLOR[0]=RGB(255,0,0)
30 DRAW.TEXT 0,160,120,"Hello!"
40 GOTO 40
```

What do all of these alignment options do – there are nine combinations. Here's a program to show you how the @FONT.HALIGN and @FONT.VALIGN system variables control the relative text positioning. First we draw a grid using DRAW.LINE commands, then we DRAW.TEXT at the grid intersection points with the various font alignment options to show how they affect the text rendering:

```
5 REM Test font HALIGN and VALIGN
10 @ANSI.ENABLE=0
20 REM Draw grid
30 DRAW.LINE 10,0,10,239,RGB(255,0,0)
40 DRAW.LINE 160,0,160,239,RGB(255,0,0)
50 DRAW.LINE 309,0,309,239,RGB(255,0,0)
60 DRAW.LINE 0,10,319,10,RGB(255,0,0)
70 DRAW.LINE 0,120,319,120,RGB(255,0,0)
80 DRAW.LINE 0,229,319,229,RGB(255,0,0)
90 REM Draw aligned text
100 @FONT.HALIGN[0]=0:@FONT.VALIGN[0]=0:DRAW.TEXT 0,10,10,"LEFT,BOT"
110 @FONT.HALIGN[0]=1:@FONT.VALIGN[0]=0:DRAW.TEXT 0,160,10,"MID,BOT"
120 @FONT.HALIGN[0]=2:@FONT.VALIGN[0]=0:DRAW.TEXT 0,309,10,"RIGHT,BOT"
130 @FONT.HALIGN[0]=0:@FONT.VALIGN[0]=1:DRAW.TEXT 0,10,120,"LEFT,MID"
140 @FONT.HALIGN[0]=1:@FONT.VALIGN[0]=1:DRAW.TEXT 0,160,120,"MID,MID"
150 @FONT.HALIGN[0]=2:@FONT.VALIGN[0]=1:DRAW.TEXT 0,309,120,"RIGHT,MID"
160 @FONT.HALIGN[0]=0:@FONT.VALIGN[0]=2:DRAW.TEXT 0,10,229,"LEFT,TOP"
170 @FONT.HALIGN[0]=1:@FONT.VALIGN[0]=2:DRAW.TEXT 0,160,229,"MID,TOP"
180 @FONT.HALIGN[0]=2:@FONT.VALIGN[0]=2:DRAW.TEXT 0,309,229,"RIGHT,TOP"
190 GOTO 190
```



Drawing Images

Using all of the DRAWing command you can construct pretty line art displays. But how do we draw images – like photos or Photoshop generated graphics?

There is a DRAW.BITMAP command that will allow you to do this. While it is simple to use there are a few caveats:

1. The image to be drawn on the screen must be a Windows .BMP bitmap file.
2. The image must be resident in the display's memory by being a Resource in the Resource table.

To quickly try this command we'll use one of the built-in image resources. Use a RESOURCES.LIST command with a wildcard name pattern to list all of the image resources:

```
resources.list *.bmp
ButtonK.bmp  1,610 builtin
ButtonK2.bmp 2,050 builtin
ButtonK4.bmp 3,018 builtin
ButtonM.bmp  2,618 builtin
Ready
```

The names suggest that these are images of the buttons that the display uses for the Configuration Settings screen keyboard. They have to be 'built-in' so that they are always there – even when no other resources are loaded.

So let's draw one of the button images in the center of the screen. As before here's the command format from the Reference:

DRAW.BITMAP x, y, imageResourceName		
Renders the bitmap resource on the current drawing @SURFACE at x, y	x	image lower-left x coordinate (0 - 319)
	y	image lower-left y coordinate (0 - 240)
	imageResourceName	the string name of the loaded image resource to draw

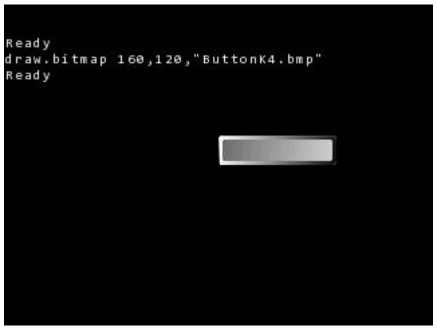
Type the following commands – first let's clear the screen:

```
draw.fill 0,0,320,240,RGB(0,0,0)
```

This leaves the Ready prompt and blinking cursor at the top of the screen – OK for now. Now let's draw the ButtonK4.bmp image at the center of the screen. Type the following command:

```
draw.bitmap 160,120,"ButtonK4.bmp"
```

Your screen should now look like:



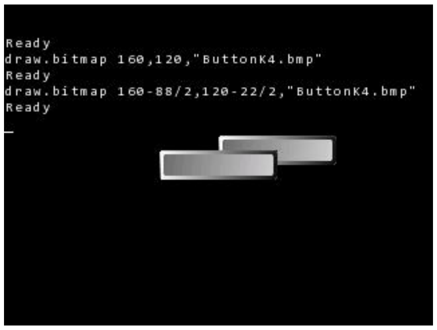
```
Ready
draw.bitmap 160,120,"ButtonK4.bmp"
Ready
```

Surprise! The image looks like the Configuration Settings screen keypad spacebar – without any colors.

Why isn't it shown in the center of the screen at 160, 120? This is because the DRAW.BITMAP coordinates reference where the lower-left corner of the bitmap image will be drawn. The ButtonK4.bmp happens to be 88 pixels wide by 22 pixels tall. So we could center it by subtracting half of these values from the x and y coordinates. Let's try moving it - type the following command:

```
    draw.bitmap 160-88/2,120-22/2,"ButtonK4.bmp"
```

Your screen should now look like:



```
Ready
draw.bitmap 160,120,"ButtonK4.bmp"
Ready
draw.bitmap 160-88/2,120-22/2,"ButtonK4.bmp"
Ready
_
```

Notice that it is centered now, and also that it was drawn on top of what was already there – like stacking photos. The effect of all of the DRAW commands is cumulative – each one builds upon the resulting image.

There are additional variations on the DRAW.BITMAP command that support transparency and indexing. These are outlined in the ACS Basic Reference section of this manual.

Rules for Drawing

1. DRAWing commands use x (horizontal) and y (vertical) coordinates to control where they affect the screen.
2. Graphic DRAWing commands specify the color to use with a 16-bit additive RGB color model – expressed with a signed integer value or using the RGB(red,green,blue) function.
3. The DRAW.POLYGON commands require arrays of the x and y coordinates of the polygon vertices. The arrays must be pre-DIMensioned and loaded with the vertex coordinates before the polygon can be drawn.
4. The DRAW.TEXT command uses embedded font file (.EFNT) resources to render the text on the screen. The style, color, alignment and transparency of the drawn text are controlled by referring to a configured entry in the FONTS table.
5. The DRAW.BITMAP commands render Windows .BMP image resources on the screen. The x and y coordinates refer to the placement of the lower-left corner of the image resource on the screen.
6. Drawing is cumulative – each successive DRAW command builds on the results of prior commands.

Making Things Move

So we know how to make static screens using the DRAW commands. How do we make something that moves?

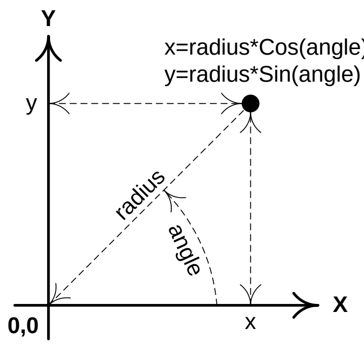
Let's draw a simple old-style sonar screen – like the ones used on submarines. It consisted of a circle and a sweeping line that rotated from the center. We know how to clear the screen, draw the circle and line, but how do we make the line rotate around the circle? We know the circle's center coordinate would be the line's starting point, but how to we dynamically compute the rotating line's end coordinate? For that we need to use some mathematics.

Warning – Mathematics

In mathematics, the trigonometric functions (also called circular functions) are functions of an angle. They are used to relate the angles of a triangle to the lengths of the sides of a triangle.

The most familiar trigonometric functions are the sine and cosine. In the context of the standard unit circle with radius 1, where a triangle is formed by a ray originating at the origin and making some angle with the x-axis, the sine of the angle gives the length of the y-component (rise) of the triangle and the cosine gives the length of the x-component (run). Trigonometric functions are commonly defined as ratios of two sides of a right triangle containing the angle, and can equivalently be defined as the lengths of various line segments from a unit circle.

Here's a drawing showing this relationship:



As we saw earlier in the section on functions, the ACS Color Basic has both SIN( ) and COS( ) functions. These trig functions take an angle argument and return the unit circle ratios – up scaled by 1024 since this Basic only provides integer math.

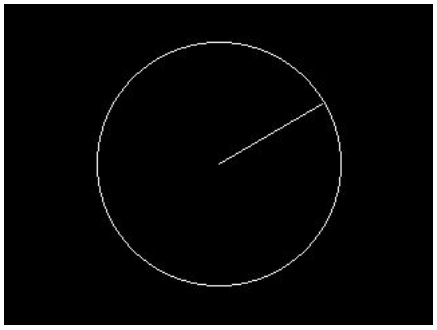
In order to perform the radius multiplication and subsequent down scaling by 1024 the MULDIV( ) function is used. This function multiplies the first two arguments as 32-bit integers and then divides by the third argument to return a 16-bit result.

We now have the pieces we need to draw our sweeping sonar screen. To rotate the line we un-draw it at its old angle then redraw it at the new angle as we sweep the angle through 360 degrees.

Here's the program:

```
10 REM Sonar Sweep
20 @ANSI.ENABLE=0:@BACKLIGHT=1
30 centerX = 160 : centerY = 120 : radius = 90
40 DRAW.FILL 0,0,320,240,RGB(0,0,0)
50 DRAW.CIRCLE centerX, centerY, radius, RGB(255,255,255)
60 FOR angle = 0 TO 359
70 lineX = MULDIV(radius-1, COS(angle), 1024)
80 lineY = MULDIV(radius-1, SIN(angle), 1024)
90 DRAW.LINE centerX, centerY, centerX + oldLineX, centerY + oldLineY, RGB(0,0,0)
100 DRAW.LINE centerX, centerY, centerX + lineX, centerY + lineY, RGB(255,255,255)
110 oldLineX = lineX : oldLineY = lineY
120 NEXT angle
130 GOTO 60
Ready
```

And here's what your display should look like – with the sweep line rotating counter-clockwise around the circle origin:

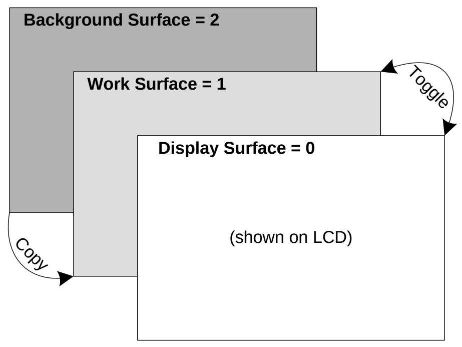


Moving Smarter

While this program works, there is a lot of flickering of the sweep line and we have to keep track of the line's old position so that we can un-draw it before redrawing it in its new position. There is a better way using an approach called ping-pong or double buffering.

With double buffering, drawing alternates between two different surfaces – one is shown while drawing on the other, then they are switched and the process repeats.

The Color LCD actually has three drawing surfaces to support this approach: Display, Work and Background:



The Background surface provides a content area that can be seldom drawn, and copied to the Work surface before additional drawing is done on top.

The Work surface is where most drawing is done before being swapped with the Display surface.

The Display surface is the one that is currently being shown on the LCD. This is the one that we have been drawing on up to now.

All of the DRAW commands operate on the current drawing surface which is set using a system variable - @SURFACE. The DRAW.TOGGLE command is used to swap the work and display surfaces – they aren't copied, the work surface becomes the display and the old display becomes the work surface.

In order to implement double buffering and avoid flickering as the display is updated we need to change our program to use the following sequence:

Double-Buffering Sequence

1. Switch to the Background draw surface.
2. Draw any required static background content.
3. Switch to the Work surface.
4. Copy in the content from the Background surface to the Work surface and draw any dynamic content on top of it.
5. Toggle the Work and Display surfaces.
6. Repeat from step 4.

Let’s modify the sonar sweep program to use this double buffering approach.

Here’s the DRAW.COPY command format from the reference:

DRAW.COPY src,sx,sy,dx,dy,w,h		
Copies the width x height bitmap from the src surface @ sx,sy on the current drawing @SURFACE at dx, dy	src	source surface number (0, 1, 2)
	sx	source lower-left x coordinate (0 – 319)
	sy	source lower-left y coordinate (0 – 240)
	dx	destination lower-left x coordinate (0 – 319)
	dy	destination lower-left y coordinate (0 – 240)
	w	width of area to copy (1 – 320)
	h	height of area to copy (1 – 240)

First we’ll select the background surface, clear it and draw in the circle – which is static – it doesn’t change:

```
@SURFACE=2
DRAW.FILL 0,0,320,240,RGB(0,0,0)
DRAW.CIRCLE centerX, centerY, radius, RGB(255,255,255)
```

Next we’ll select the work surface:

```
@SURFACE=1
```

Then we copy the background surface area encompassing the circle and draw the sweep line on it:

```
DRAW.COPY 2, leftX, leftY, leftX, leftY, width, height
lineX = MULDIV(radius-1, COS(angle), 1024)
lineY = MULDIV(radius-1, SIN(angle), 1024)
DRAW.LINE centerX, centerY, centerX + lineX, centerY + lineY,
RGB(255,255,255)
```

Finally we toggle the work and display surfaces then rinse and repeat.

```
DRAW.TOGGLE
```

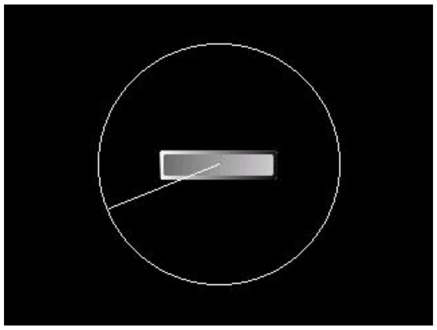
Here’s the complete modified sonar sweep program that now uses the double buffering approach – run it and notice that the flickering is gone:

```
10 REM Sonar Sweep
20 @ANSI.ENABLE=0 : @BACKLIGHT=1
30 centerX = 160 : centerY = 120 : radius = 90
40 leftX=centerX-(radius+1) : leftY=centerY-(radius+1) : width=(radius+1)*2 : height=(radius+1)*2
50 @SURFACE=2
60 DRAW.FILL 0,0,320,240,RGB(0,0,0)
70 DRAW.CIRCLE centerX, centerY, radius, RGB(255,255,255)
80 @SURFACE=1
90 FOR angle = 0 TO 359
100 DRAW.COPY 2, leftX, leftY, leftX, leftY, width, height
110 lineX = MULDIV(radius-1, COS(angle), 1024)
120 lineY = MULDIV(radius-1, SIN(angle), 1024)
130 DRAW.LINE centerX, centerY, centerX + lineX, centerY + lineY, RGB(255,255,255)
140 DRAW.TOGGLE
150 NEXT angle
160 GOTO 90
```

Notice that the circle is only drawn once – in fact it could be a bitmap image. Add the ButtonK4.bmp image into the background screen:

```
75 draw.bitmap centerX-88/2,centerY-22/2,"ButtonK4.bmp"
```

Save and run it again. You should see the sweep line rotating on top of the button:



Changing Direction

The last example ran the animation at full speed – there was no delay between screen updates and the angle was incremented through an entire circle of 360 degrees in one degree steps.

What if you wanted to simulate the second hand of a clock? A few changes would be necessary; the motion should only occur once per second, the rotation would have to move clockwise instead of counter-clockwise, and since there are 60 seconds around the face of a clock the motion would have to advance:

$$\frac{360\,degrees}{60\,seconds} = 6\frac{degrees}{second}$$

Also – zero seconds is straight up to the 12 o'clock position – which is not the zero angle position but 90 degrees.

Let's start by tackling the non-zero angle for zero seconds and the clockwise motion. Remember from the earlier mathematics section that the angle advances counter-clockwise as the angle increases. If you change line 90 in the previous program to STEP the FOR/NEXT loop from 359 to 0 and run the program you will see that the motion is now in the clockwise direction. If you change the STEP value from -1 to -6 the motion is faster, but still in a clockwise direction.

So how can we translate the seconds to the required negative angle with the 90 degree offset? Let's start by offsetting the zero angle from the 360 degree position by 90 degrees – 360 + 90 = 450 degrees. From the previous equation and results we know that we have to subtract 6 degrees per second:

```
angle = (360 + 90) - (second * 6)
```

This equation yields angle = 450 when second = 0, and angle = 96 when second = 59. We need a way to limit the range of the computed angle to 0 – 359 degrees for use with the SIN( ) and COS( ) functions.

Enter the modulus operator: %. Integer division truncates (discards) the fractional part of the result. The modulus operator divides the value to its left by the value on its right and returns the remainder. When both values are equal or integer multiples, the remainder of the division is zero. This can be evaluated as:

x % y = (x – ((x / y) * y))

Try computing the modulus of 4 for various values:

0 % 4	= (0 – ((0/4) * 4))	= 0
1 % 4	= (1 – ((1/4) * 4))	= 1
2 % 4	= (2 – ((2/4) * 4))	= 2
3 % 4	= (3 – ((3/4) * 4))	= 3
4 % 4	= (4 – ((4/4) * 4))	= 0
5 % 4	= (5 – ((5/4) * 4))	= 1
6 % 4	= (6 – ((6/4) * 4))	= 2

You can see that the modulus ranges from zero when the arguments are equal to one less than the modulus. So taking the modulus 360 of the angle computation returns an angle between 0 and 359:

```
angle = ((360 + 90) - (second * 6) % 360)
```

If you now change lines 90 and 150 to count seconds from 0 to 59 and add line 95 to compute the angle you will see the motion being driven counter-clockwise in 6 degree increments as the seconds range from 0 to 59.

Rules for Making Things Move

1. To avoid flickering when moving graphics by DRAWing, use the double-buffering sequence.
2. You can compute the X and Y coordinates for DRAWing circular motion using the trig functions SIN( ) and COS( ) and scale the results using the MULDIV( ) function.
3. Incrementing angles rotate counter-clockwise, decrementing angles rotate clockwise.
4. The modulus operator % returns the remainder of the division of the two arguments.

Staging an Event

We could add a 1 second delay using a @TIMER system variable and increment the second variable from zero through fifty-nine every time that we read @TIMER is zero. There is another way that will allow our program to do other things instead of simply spinning in a loop waiting on a timer. Welcome to Events.

Events are changes in your program's execution that can happen in between each statement. The linear execution of your program's statements is interrupted to process the event and then execution resumes from where your program was interrupted. Events are associated with and triggered by some system variables and can occur as a result of hardware or software actions.

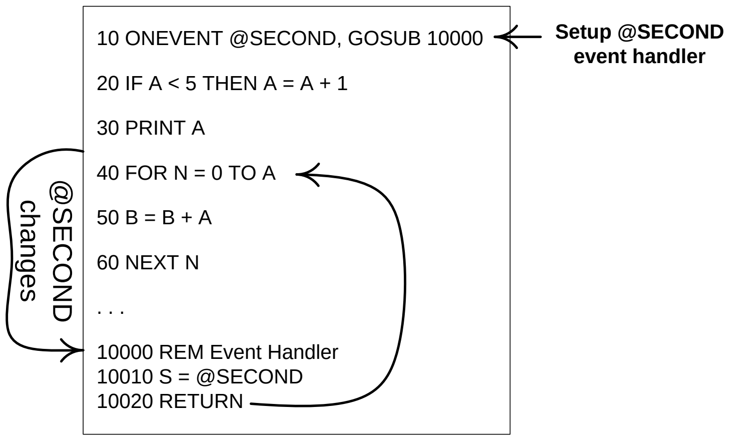
You control how system variable events are processed in your program by associating them with an event handler. The event handler is a subroutine you write that will be called when the event is signaled by changes in the associated system variable.

The event handler is configured or setup using the ACS Color Basic's ONEVENT command:

```
ONEVENT @systemvariable, GOSUB linenumber
```

This command records the handler subroutine line number in an event table for the specified system variable entry. Changes to system variables that cause events set a flag on their entry in the event table.

In between processing each program statement the Color LCD scans the event table looking for flagged entries. When a flagged entry is found the display performs the GOSUB to the recorded event handler line number. When the event handler subroutine returns, the display clears the flagged entry.



```
10 ONEVENT @SECOND, GOSUB 10000
20 IF A < 5 THEN A = A + 1
30 PRINT A
40 FOR N = 0 TO A
50 B = B + A
60 NEXT N
. . .
10000 REM Event Handler
10010 S = @SECOND
10020 RETURN
```

Events are prioritized – an executing event handler can only be interrupted by a higher priority event.

It is important to remember that all variables are shared between all parts of your program. If your program is performing some long calculation using a variable and you change that variable in an event handler your program may get confused or generate incorrect results since the event handler can potentially execute anywhere in your program after it is configured.

So let's add event processing to our second hand program. There is a system variable @SECOND that is updated once a second from the Color LCD Real Time Clock hardware. It signals an event when this happens. We will turn the program lines 95 – 160 into a @SECOND event handler.

Edit line 95 to change the second to the @SECOND system variable in the angle calculation:

```
95 angle = ((360 + 90) - @SECOND * 6) % 360
```

Change line 150 to a RETURN statement and remove line 160. Now add line 85 to configure the @SECOND event handler to call the new subroutine at line 95:

```
85 ONEVENT @SECOND,GOSUB 95
```

And, just to show that your program is capable of doing other things while updating the screen once a second replace line 90 with:

```
90 a = a + 1 : PRINT a : GOTO 90
```

Your program should now look like:

```
 10 REM Sonar Sweep
20 @ANSI.ENABLE=0:@BACKLIGHT=1
30 centerX = 160 : centerY = 120 : radius = 90
40 leftX = centerX-(radius+1) : leftY = centerY-(radius+1) : width = (radius+1)*2 : height = radius+1)*2
50 @SURFACE=2
60 DRAW.FILL 0,0,320,240,RGB(0,0,0)
70 DRAW.CIRCLE centerX, centerY, radius, RGB(255,255,255)
75 DRAW.BITMAP centerX-88/2,centerY-22/2,"ButtonK4.bmp"
80 @SURFACE=1
85 ONEVENT @SECOND,GOSUB 95
90 a = a + 1 : PRINT a : GOTO 90
95 angle = ((360 + 90) - @SECOND * 6) % 360
100 DRAW.COPY 2,leftX,leftY,leftX,leftY,width,height
110 lineX = MULDIV(radius-1, COS(angle), 1024)
120 lineY = MULDIV(radius-1, SIN(angle), 1024)
130 DRAW.LINE centerX, centerY, centerX + lineX, centerY + lineY, RGB(255,255,255)
140 DRAW.TOGGLE
150 RETURN
```

If you run this program, you will see the line rotating like a second hand – clockwise, 6 degrees every second. At the same time, your program is PRINTing an incrementing number.

Rules for Events

1. Events are interruptions to your program's normal flow of execution caused by hardware or software changes to system variables.
2. To process events you create an association between a system variable and a subroutine in your program using the ONEVENT command. The subroutine will be executed when the event is signaled by a change to the system variable. The event is cleared by the RETURN from the event.
3. You can disable further event handling for a system variable by executing an ONEVENT command with a line number of zero.
4. Variables are shared between your mainline program and any event handlers. Care should be exercised to ensure that inadvertent changes to variables in an event handler don't affect your main program logic.
5. Events are prioritized - only higher priority events able to interrupt an executing event handler.
6. Event handlers should be short so that lower priority events have a chance to occur.

Taking Control with the Screen Framework

By now you can probably see how you could use the DRAWing commands to construct some pretty fancy screens with animation.

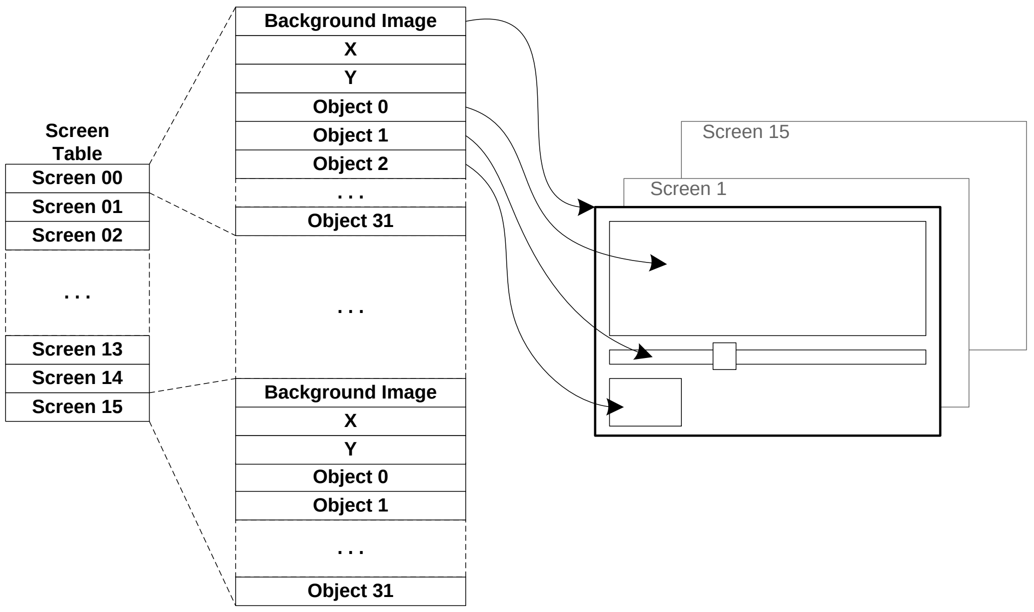
There is another method available to provide multiple screens, each with multiple controls – controls that provide a higher level of functionality and take care of drawing themselves and interacting with the user. This is accomplished using a built-in Screens Framework that ACS Color Basic has the commands and system variables to interact with.

There's a Table of Screens

Earlier we saw how embedded fonts and bitmap images were accessed using Resources and the Font table. The Screens Framework adds two more tables to the mix; a Screen table and a Scheme table.

The Screen table holds information about how to draw each Screen entry and what controls (Screen Objects) each Screen uses. The Screen table currently holds sixteen Screen entries, and each Screen entry currently holds up to thirty-two Screen Object entries.

Screen Object entries define what type of object (button, slider, gauge, label, etc.), how it is rendered (images, scheme, text, options), where it is positioned in relationship to the containing screen, what 'value' it has and other information used to interact with the user and running program. Here's a diagram:



ACS Color Basic has system variables to access the elements of the Screen table and commands to persist and 'navigate' the table – saving and loading the table and what screen entry to change to, push to or pop from. Screens are not stored fully rendered in memory but are 'constructed' (drawn) when navigated to.

Scheming Controls

The Schemes table holds information about how to draw and colorize images and display text.

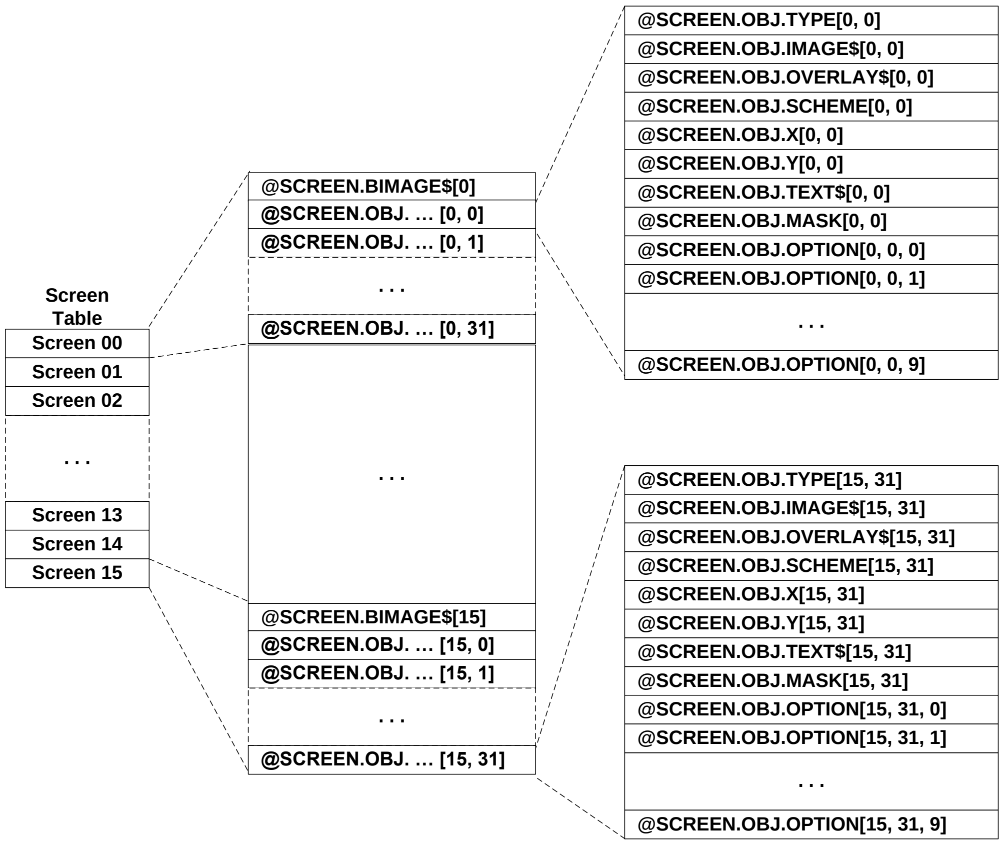
Screen Objects can reference Scheme table entries to control how they are drawn and colored as well as how any text that they display is rendered. Scheme table entries are usually used in pairs with the lower-numbered entry used to render an un-touched control and the higher-numbered entry used to render a control that is touched. In the previous section we drew a built-in keypad button bitmap that was drawn as grayscale – only shades of black to white were used. The Screens Framework's use of schemes allows these buttons and their text to be colorized – and the color to change as the buttons are pressed and released. This provides an easy way to accomplish a common look and feel for screen objects with the ability to easily change their appearance from one place. We'll examine schemes a little later on.

Controlling Screens and Objects

The Screen table elements are accessed using indexed system variables - the @SCREEN prefix is followed by one or more 'selectors' using the dot notation to select which entry's element is being accessed followed by the index to select the entry number in square brackets.

Accessing a Screen's object element requires a dual index to select the screen number then the object number. Finally, accessing a Screen Object's option element requires a triple index to select the screen number, object number and option number.

Here's a diagram showing the system variables and indices:



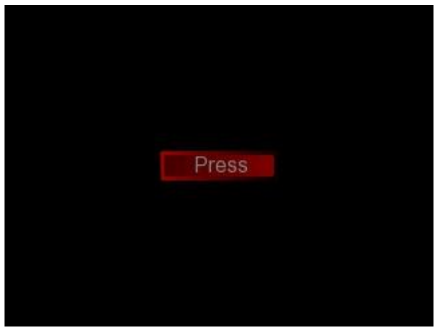
Touch Button Control

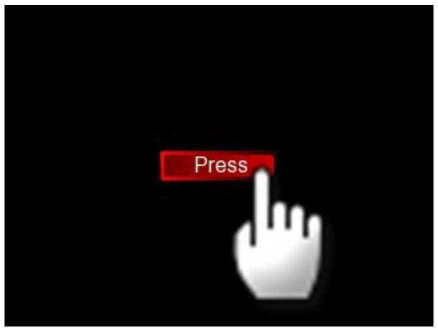
Let's try drawing a simple screen with no background image and a single button labeled 'Press'. Type in the following short program:

```
10 REM Button Demo
15 @ANSI.ENABLE=0 : REM Disable ANSI text
20 @SCREEN.OBJ.TYPE[0,0]=2 : REM Set screen 0 object 0 to Button type
25 @SCREEN.OBJ.SCHEME[0,0]=0 : REM Set button's scheme pair to 0,1
30 @SCREEN.OBJ.X[0,0]=116 : @SCREEN.OBJ.Y[0,0]=109 : REM Set button's location on screen
35 @SCREEN.OBJ.IMAGE$[0,0]="ButtonK4.bmp" : REM Set button's image
40 @SCREEN.OBJ.TEXT$[0,0]="Press" : REM Set button's text
45 SCREENS.CHANGETO 0 : REM Activate screen 0
999 GOTO 999
```

This program initializes the first screen object zero of the first screen table entry zero to position a button drawn using the built-in ButtonK4.bmp resource at the center of the screen. The gray button is labeled with the text ‘Press’ and is colorized using scheme table entry zero. The Screen Framework is then started by the navigation to this screen using the SCREENS.CHANGETO command.

Your display should look like the following – note how pressing the button changes its color shading automatically using the second entry of the scheme pair:





So how does your program ‘know’ when the user has pressed the button? There is a system variable that gets updated whenever a screen object changes state:

@SCREEN.OBJ.EVENT - This system variable reflects the last screen object event. There are currently three screen object events:

Value of @SCREEN.OBJ.EVENT	Indicates Screen Object Event
0	No event
1	Screen Object was touched
2	Screen Object value has changed
3	Screen Object was un-touched (released)

So let’s change the program to watch this system variable and PRINT the event. To do this we’ll use a new command – LIF, to control the program execution.

LIF is short for Long IF. If you remember the IF statement from *Controlling Program Execution* above, the command evaluates the following condition and IF it is True (non-zero) THEN it executes the command after the THEN otherwise it skips to the next line.

The LIF command works the same way except it executes all the commands on the line after the THEN when the condition evaluates to non-zero. This is useful when you want your program to do more than one command after testing a condition and avoid having to GOTO a new group of commands on another line to do so – it can make the program easier to write, follow and understand.

Add lines 50 – 65 below to the program and run it. These new lines test the value of the @SCREEN.OBJ.EVENT system variable and PRINT a message identifying the event. The event is then ‘cleared’ by setting it to zero and loops back to the top of the stack of LIF statements.

Try pressing and releasing the button and see what it printed:

```
10 REM Button Demo
15 @ANSI.ENABLE=0 : REM Disable ANSI text
20 @SCREEN.OBJ.TYPE[0,0]=2 : REM Set screen 0 object 0 type to Button
25 @SCREEN.OBJ.SCHEME[0,0]=0 : REM Set button's scheme pair to 0,1
30 @SCREEN.OBJ.X[0,0]=116 : @SCREEN.OBJ.Y[0,0]=109 : REM Set button's location on screen
35 @SCREEN.OBJ.IMAGE$[0,0]="ButtonK4.bmp" : REM Set button's image
40 @SCREEN.OBJ.TEXT$[0,0]="Press" : REM Set button's text
45 SCREENS.CHANGETO 0 : REM Activate screen 0
50 LIF @SCREEN.OBJ.EVENT = 1 THEN PRINT "Pressed" : @SCREEN.OBJ.EVENT = 0 : GOTO 50
55 LIF @SCREEN.OBJ.EVENT = 2 THEN PRINT "Value" : @SCREEN.OBJ.EVENT = 0 : GOTO 50
60 LIF @SCREEN.OBJ.EVENT = 3 THEN PRINT "Released" : @SCREEN.OBJ.EVENT = 0 : GOTO 50
65 GOTO 50
999 GOTO 999
```

```
run
Pressed
Value
Released
Value
. . .
Pressed
Value
Released
Value
```

Note the sequence of screen object events; Pressed, Value, Released, Value. A button only has two values; 0 and 1, but other controls like a slider can have several values that change between being pressed then released.

Multiple Objects

So how does your program handle events for multiple objects on a screen? There is another system variable that gets updated at the same time as @SCREEN.OBJ.EVENT with the number of the object that caused the event:

@SCREEN.OBJ.# - This read-only system variable reflects the screen object number of the last screen object event.

The previous screen framework example for a single button polled the @SCREEN.OBJ.EVENT using the multiple LIF statements in a loop. However, the @SCREEN.OBJ.EVENT is event capable and, in an associated event handler, you can use the @SCREEN.OBJ.# system variable to identify what object the event is being generated from.

To simplify this processing we introduce a new ON command. The ON command provides the ability to change the program’s execution using the value of an expression in a single statement instead of testing for each condition with multiple IF or LIF statements. It can make a program smaller and easier to read and understand.

The ON command is followed by the expression to evaluate, then the execution directive which can be either GOTO or GOSUB, and then a list of line numbers for each value of the expression. The destination line number is selected from the list by the value of the expression.

```
ON Expr, GOTO lineIfExpr=0, lineIfExpr=1, … , lineIfExpr=n

ON Expr, GOSUB lineIfExpr=0, lineIfExpr=1, … , lineIfExpr=n
```

If the expression evaluates to zero, the first line number in the list is used for the GOTO / GOSUB, an expression value of one selects the next line number, two the next and so on.

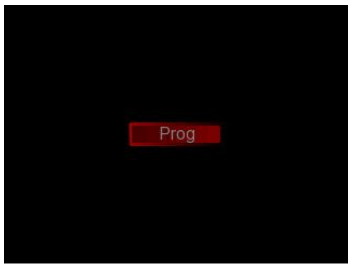
If the expression evaluates to a negative value, or to a value beyond the number of line numbers in the list, execution continues with the program statement following the ON command; this is the default case if there isn’t a line number to jump to for an expression value. If the selected line number in the list is zero, execution continues with the program statement following the ON command; this allows certain expression values within the list of line numbers to be handled as defaults.

So let’s add an event handler for the @SCREEN.OBJ.EVENT. We’ll use the ON command to help process the event. We’ll add a label object to the screen. The event handler will update the label object’s .TEXT$ to display the button’s last state; Pressed or Released. Here’s the revised code:

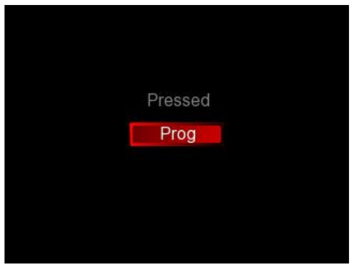
```
10 REM Button Demo
15 @ANSI.ENABLE=0 : REM Disable ANSI text
20 @SCREEN.OBJ.TYPE[0,0]=2 : REM Set screen 0 object 0 to Button
25 @SCREEN.OBJ.SCHEME[0,0]=0 : REM Set button's scheme pair to 0,1
30 @SCREEN.OBJ.X[0,0]=116 : @SCREEN.OBJ.Y[0,0]=109 : REM Set button's location on screen
35 @SCREEN.OBJ.IMAGE$[0,0]="ButtonK4.bmp" : REM Set button's image
40 @SCREEN.OBJ.TEXT$[0,0]="Prog" : REM Set button's text
45 @SCREEN.OBJ.TYPE[0,1]=6 : REM set screen 0 object 1 to Label
50 @SCREEN.OBJ.SCHEME[0,1]=0 : REM set label's scheme pair to 0,1
55 @SCREEN.OBJ.X[0,1]=160 : @SCREEN.OBJ.Y[0,1]=151 : REM set label's location on screen
60 @SCREEN.OBJ.OPTION[0,1,0]=88 : REM override label width
```

```
65 ONEVENT @SCREEN.OBJ.EVENT,GOSUB 1000
70 SCREENS.CHANGETO 0 : REM activate screen 0
75 GOTO 75
1000 REM screen object event handler
1005 IF @SCREEN.OBJ.#=0 THEN ON @SCREEN.OBJ.EVENT-1, GOTO 1015,0,1020
1010 RETURN
1015 @SCREEN.OBJ.TEXT$[0,1]="Pressed" : RETURN
1020 @SCREEN.OBJ.TEXT$[0,1]="Released" : RETURN
```

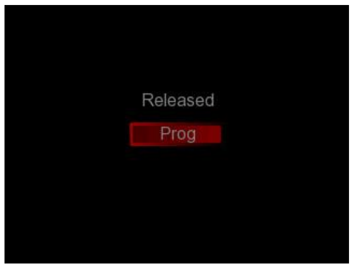
And now, when we run it and press the button the label above the button is updated to show the button's last state:



App started



Button pressed



Button released

Diving Into the Screen Framework

So how does this Screen Framework actually work? It uses messages and the multiple drawing surfaces system to perform its magic.

When the screen framework is started by the initial SCREENS.CHANGETO command, the screen is constructed using the following sequence of steps:

1) If there is a screen background image .BMP resource specified it is drawn onto the background surface, if not the background surface is filled with black.
2) Each screen object is sent a message to render their background information.
3) The background surface is then copied to the work surface.
4) Each screen object is sent a message to render their working information.
5) The work and display surfaces are then toggled so that what was the working surface is now the display surface and is being shown on the LCD.
6) Each screen object is sent a message to render any additional dynamic content on the displayed surface.

After the screen is constructed, a construction event is signaled via the @SCREEN.EVENT system variable, with the @SCREEN.# variable identifying what screen number the event is being generated from. This allows the Basic program to initialize any variables, start timers or perform other functions specific to or required by the activating screen.

If a new screen is navigated to by a .CHANGETO or .PUSH or .POP command while a current screen is being displayed, a destruction event is signaled via the @SCREEN.EVENT variable and then the new screen is constructed. This allows the Basic program to stop timers or disable other functions that were in use by the previous screen.

When the user touches the screen a message containing the touch state; press, move or release, along with the screen coordinates is sent to each object on the screen. Each object determines whether it is affected by the touch event, possibly updating its state and updating its appearance on the screen. If the object determines is has been touched or its .VALUE has changed an event is signaled via the @SCREEN.OBJ.EVENT system variable, with the @SCREEN.OBJ.# variable identifying what screen object number the event is being generated from.

The Basic program can DRAW on the display surface after the screen constructed event has occurred. Any drawing done on the display surface between the SCREEN navigation command and the screen constructed event would appear to be drawn on the work surface after step 3 above occurs. Double buffer or ping-pong drawing should not be performed as it will interfere with the Screen Framework's operation.

Screen object's .VALUEs can be changed by the program to cause the object to change its appearance on the screen. These changes should be made after the object's containing screen has been constructed as determined by the construction event. Changes before the occurrence of the screen construction event may be overwritten or discarded until the screen (and its objects) have been constructed.

Using the screen framework and controls can simplify a Color Basic graphic application as the screens and objects manage the drawing surfaces and render themselves in response to system variable changes and user touch interaction.

Rules for the Screen Framework

1. The Screen Framework is comprised of multiple system variable tables that specify what objects (controls) are located where on each screen and how they are displayed.
2. The Screen Framework is started by the first SCREENS.CHANGETO or SCREENS.PUSHTO command.
3. Screens are not stored fully rendered but are constructed (drawn) when navigated to.
4. Screens and screen objects signal changes by using events.
5. Screens can be DRAWn on after they are constructed but not by using the double-buffering technique.

to be continued…

ACS Basic Reference

ACS Basic is an integer, microcomputer basic designed for simple control applications.

ACS Basic executes programs consisting of one or more statements. Statements consist of an optional line number followed by reserved keyword commands specifying operations for Basic to perform followed by required and / or optional arguments.

Statements that begin with a line number are entered and held, sorted by line number, until Basic is commanded to execute them. This is called the **Program** mode of operation. Statements entered without a line number are evaluated and executed immediately. This is called the **Direct** mode of operation. Some keyword commands are **Direct** mode only and may not appear in a program. Some keyword commands are **Program** mode only and may not be evaluated and executed immediately after being typed in. These limitations are listed in the keyword command definitions below.

Programs

In ACS Basic a Program consists of one or more program lines. Each program line consists of a line number followed by one or more statements. Valid line numbers range from 1 to 2,147,483,647. Multiple statements in a program line must be separated by colons (":"). Program lines that are entered without a line number are executed directly. Only certain statements may be executed directly. When ACS Basic is awaiting statement or program line entry it issues a READY prompt via the serial port.

```
Ready
dir *.bas
TEVENT.BAS          250 A      11-09-2058 14:30:10
PROGRAM1.BAS         55 A      11-09-2058 15:52:44
SOUNDS.BAS          248 A      01-01-1980 00:00:00
TEST.BAS             63 A      01-01-1980 00:00:00
CEVENTS.BAS         144 A      01-01-1980 00:00:00
PROGRAM2.BAS         47 A      11-09-2058 15:58:14
ONGOTO.BAS          253 A      05-08-2052 14:35:54
ONGOSUB.BAS         272 A      11-09-2058 14:45:08
TIMER.BAS           185 A      11-15-2058 15:20:26
CHIMES.BAS          884 A      09-07-2021 16:55:10
LCDDEMO.BAS        2143 A      11-13-2020 18:36:26
MSGTEST.BAS          78 A      11-11-2020 16:15:32
-----------------------
                   12 files
                    0 directories
Ready
```

Programs may be entered a line at a time by a stream of characters via the serial port, or by loading from a file off of an optional microSD card. When entered via the serial port, a program line will replace any matching program line, and entering a line number only will delete the corresponding program line. Entered program lines are limited to 255 characters of length.

```
10 PRINT "This is a Test"
20 STOP
list
10 PRINT "This is a Test"
20 STOP
Ready
20
list
10 PRINT "This is a Test"
Ready
run
This is a Test
Ready
print "This is also a Test"
This is also a Test
Ready
```

ACS strongly recommends developing Basic programs interactively via a connected terminal / computer or optional PS2 keyboard so that error messages can be viewed and the program operation can be refined quickly – otherwise the program may silently stop running leaving no clue as to what has happened.

Program lines may be viewed with the **LIST** statement. All program lines may be cleared with the **NEW** statement. Program execution is started using the **RUN** statement. Upon power-up, ACS Basic clears the program memory and awaits statement or program line entry via the serial port.

Program lines may be edited via a connected ANSI terminal (or computer with ANSI terminal emulation) with the **EDIT** statement. (See the **EDIT** keyword command definition below for more information.)

Entering an Escape character (0x1B) twice in succession via the serial port while a program is running will cause termination of the program and ACS Basic will output a message then await further statement or program line entry via the serial port. If the program is awaiting input by executing an **INPUT** statement a trailing carriage return may be necessary to terminate the **INPUT** before the Escape sequence is seen.

```
new
Ready
10 for i=1 to 10
20 print i
30 delay(10)
40 next i
list
10 FOR i=1 TO 10
20 PRINT i
30 DELAY(10)
40 NEXT i
Ready
run
 1
 2
 3
 4    <- Escape key pressed twice here
ESC at line 20
Ready
```

Variables

ACS Color Basic has four types of variables: **32-bit Integer Numeric**, **32-bit Integer Numeric Arrays, unsigned 8-bit character Strings** and **unsigned 8-bit character String Arrays**.

Variable names *are* case sensitive. The may contain letters, numbers and underscore but they must start with a letter. They can be up to 32 characters long.

Numeric variables can assume the integer values **(–2,147,483,648 ≤ variable ≤ +2,147,483,647)**.

Character Strings are limited to **255 characters** in length.

Variable arrays are indexed with an array subscript enclosed in square brackets **[ ]** and must be **DIM**ensioned before they are used.

The number of variables is limited only by the available memory. Shorter variable names execute slightly faster.

System Variables

ACS Basic also has built-in system variables. System variables are denoted by a '**@**' character as the first character of the variable name. The system variable names are 'tokenized' when entered to save program memory and speed program execution: for example the system variable **@SECOND** would be tokenized to two bytes instead of seven bytes.

System variables **may not** be assigned a value by appearing in an **FOR, DIM, INPUT, READ, FINPUT #N** or **FREAD #N** statement. Some system variables are read-only and may not appear on the left hand side of a **LET** assignment statement.

Some system variables have ***Events*** associated with them and may be referenced in **ONEVENT, SIGNAL** and **WAIT** statements. See the description for the individual system variables and the ***Events*** section below for more information.

@TIMER[x]

The **@TIMER(x)** system variables allow the ACS Basic program to measure or control time intervals. There are ten timers; permissible values for [x] are 0 through 9. Setting the variable to a non-zero value activates the timer. The value in the timer variable is decremented every 20mSEC (50 Hz) until it reaches zero. Upon reaching zero any associated event handler specified with the **ONEVENT** statement is activated.

@CONTACT[x]

The **@CONTACT[x]** system variables allow the ACS Basic program to access the optional ACS-COLOR-LCD-PS2-IO Module contacts. The module must be installed and the PS/2 Enable configuration must be set to true. There are 4 contact inputs and 4 contact outputs; permissible values for **[x]** are 0 through 3. Setting the variable to a '1' activates output contact [x]. Reading the variable returns the value from the input contact [x].

@CLOSURE[x]

The **@CLOSURE[x]** system variables allow the ACS Basic program to access the optional ACS-COLOR-LCD-PS2-IO Module contact events. The module must be installed and the PS/2 Enable configuration must be set to true. There are 4 contact inputs; permissible values for [**x**] are 0 through 3. Reading the variable returns a '1' if the input ***contact[x]*** has had a closure since last being read. Closures are 'sticky' and the program must 'clear' the closure by assigning it a zero before it can be detected again. Optionally an event handler specified with the **ONEVENT** statement may be activated upon an input closure, which automatically clears the closure.

```
10 ONEVENT @CLOSURE[0],GOSUB 100
20 ONEVENT @CLOSURE[1],GOSUB 200
30 GOTO 30
100 PRINT "contact 0 closed":RETURN
200 PRINT "contact 1 closed":RETURN
Ready
run
contact 0 closed
contact 1 closed
```

@OPENING[x]

The **@OPENING[x]** system variables allow the ACS Basic program to access the optional ACS-COLOR-LCD-PS2-IO Module contact events. The module must be installed and the PS/2 Enable configuration must be set to true. There are 4 contact inputs; permissible values for [**x**] are 0 through 3. Reading the variable returns a '1' if the input ***contact[x]*** has had an opening since last being read. Openings are 'sticky' and the program must 'clear' the opening by assigning it a zero before it can be detected again. Optionally an event handler specified with the **ONEVENT** statement may be activated upon an input opening, which automatically clears the opening.

@CAPTURE[x], @CAPTURE.CLOSURES[x], @CAPTURE.OPENINGS[x]

The **@CAPTURE[x]** system variables allow the ACS Basic program to access the optional ACS-COLOR-LCD-PS2-IO Module capture events. The module must be installed and the PS/2 Enable configuration must be set to true. There are 4 contact inputs that record capture events; permissible values for **[x]** are 0 through 3.

@PWM.PERIOD[x], @PWM.ON[x]

The **@PWM.** system variables allow the ACS Basic program to access the optional ACS-COLOR-LCD-PS2-IO Module to configure outputs to be pulse width modulated (PWM). Once configured the IO module will turn the contact output on and off for the on portion of the PWM period. There are 4 contact outputs that can be individually programmed to generate different PWM signals; permissible values for **[x]** are 0 through 3.

The values are expressed with 1 millisecond resolution – a **@PWM.PERIOD** value of 100 is equal to 100 milliseconds or 10 hertz. Setting the **@PWM.ON** value to 50 in this example would result in a 50 / 100 duty cycle of 50% of 50 milliseconds on followed by 50 milliseconds off.

A running PWM may be stopped by setting the **@PWM.PERIOD** to zero. Setting the **@CONTACT[x]** system variable to '1' for an output overrides the PWM operation for that output.

@FILE.SIZE[#N]

The **@FILE.SIZE[#N]** read-only system variable allows the ACS Basic program to determine the size in bytes of a previously opened file **#N**.

@FILE.POSITION[#N]

The **@FILE.POSITION[#N]** system variable allows the ACS Basic program to ascertain or set the position of the next file read or write operation of a previously opened file **#N**.

Files opened in read-only mode clip the set position to the size of the file. Files opened in write-only or update mode are extended if the position is set past the current size of the file.

@FEOF[#N]

The **@FEOF[#N]** system variable allows the ACS Basic program to determine when an end-of-file has occurred after an **FOPEN #N, INPUT #N, FREAD #N** or **FINPUT #N** statement. Optionally an event handler specified with the **ONEVENT** statement may be activated upon an end-of-file occurring.

@SOCKET.EVENT[#N]

The **@SOCKET.EVENT[#N]** system variable allows the ACS Basic program to determine the state of an opened streaming socket connection. Optionally an event handler specified with the **ONEVENT** statement may be activated when an **SOCKET.EVENT** occurs. The **SOCKET.EVENT** values are:

SOCKET.EVENT[#N] value	**Event**	**Description**
0	None	No event
1	Disconnect / Done	Socket disconnected, connection done
2	Failed Open	SOCKET[#N] failed to open
3	Connection Timeout	No connection to socket within @SOCKET.TIMEOUT[#N]
4	Data Sent Timeout	No send data acknowledgment within @SOCKET.TIMEOUT[#N]
5	Data Received Timeout	No received data within @SOCKET.TIMEOUT[#N]

@SOCKET.TIMEOUT[#N]

This system variable allows the ACS Basic program to control the timeout period of socket connection, send data and receive data phases.

@SECOND, @MINUTE, @HOUR, @DOW, @DATE, @MONTH, @YEAR

These system variables allow the ACS Basic program to access the Real-Time Clock/Calendar. Writing one of these variables except @SECOND stops the clock and updates the associated value. Writing to the @SECOND variable updates the value and starts the clock running. The values of these variables are updated once per second. Whenever one of the values of these variables changes, any associated event handler specified with the **ONEVENT** statement is activated. See the ***Setting the Real Time Clock*** sample program in the Examples section for more information.

@SECOND	00 ≤ **seconds** ≤ 59
@MINUTE	00 ≤ **minutes** ≤ 59
@HOUR	00 ≤ **hour** ≤ 23
@DOW	0 ≤ **day of week** ≤ 6 (read-only, 0=Sunday)
@DATE	1 ≤ **date of month** ≤ 31
@MONTH	1 ≤ **month of year** ≤ 12
@YEAR	00 ≤ **year** ≤ 99

@SOUND$

The **@SOUND$** system variable allows the ACS Basic program to queue sound files for playing. Queued sound files are played in the order that they were queued, being removed as they are played. A sound is queued by assigning the string value of the sound filename to the variable. The currently playing sound may be determined by reading the value of the variable. The queue may be flushed by assigning an empty string to the variable. When the queue becomes empty any associated event handler specified with the **ONEVENT** statement is activated. Up to 128 sounds may be queued. ***Attempting to queue a sound when the queue is full results in an "Invalid .WAV file" error.*** Queued sounds play even if the Basic program has stopped.

@BAUD

The **@BAUD** system variables allow the ACS Basic program to control the Color LCD serial port baud rate. The baud rate is set by assigning a numeric selector value to the variable. The current baud rate selector may be determined by reading the numeric value of the variable. Note that the baud rate selector is saved in non-volatile memory and is restored every time the Color LCD powers up. ***The non-volatile memory has a limited number of write cycles (~100,000) and can be worn out by excessive writes so this system variable should not be written in a loop or on every program execution. Exercise caution to avoid non-volatile memory failure. A good practice is to check the variable's value and only then write to it if it is not the desired value.***

@BAUD	Baud Rate
0	110
1	300
2	600
3	1200
4	1800
5	2400
6	3600
7	4800
8	7200
9	9600 (factory default)
10	14400
11	19200
12	28800
13	38400
14	57600
15	115200

@MSG$

This system variable is updated by receipt of a serial data stream message that is framed with the **@SOM** and **@EOM** characters which are not included in the **@MSG$**. It retains the framed message until it is read at which point the search for the next received **@SOM** begins again. It may also be cleared by assigning it a string value, which is not saved.

@MSGENABLE

This system variable controls whether the serial data stream is parsed for messages as outlined in the **@MSG$** description above. The ability to disable **@MSG$** processing is required to support the **GETCH()** function on the serial port. It defaults to 1 (enabled).

@DMX

These system variables are used to access the DMX512 functionality.

@DMX.RESET

Writing this system variable to a non-zero value resets the DMX.

@DMX.MASTER

Writing this system variable to a non-zero value enables the optional DMX as a master, controller if present. A value of zero enables sets slave, device mode.

@DMX.DATA[x]

Gets or sets the current value of channel x (0 ≤ x ≤ 511) if the optional DMX I/O module is present.

@EOT

This system variable returns 1 when any serial data sent by BASIC console operation, or PRINT or LCDx statements has finished transmitting. It can be cleared by setting it to zero, but will immediately return 1 again unless serial data is sending.

@SMTP

These system variables are used to provide events and status when sending e-mail via the Simple Mail Transfer Protocol (SMTP).

@SMTP.EVENT

This system variable reflects the last SMTP event:

@SMTP.EVENT	Event
0	None
1	Status Update
2	Connection Aborted
3	Connection Failed
4	HELO / EHLO Failed
5	AUTH LOGIN Failed
6	FROM Failed
7	TO Failed
8	CC Failed
9	DATA Failed
10	QUEUE Failed
11	SUCCESS

@SMTP.MESSAGE$

This system variable holds any text message associated with the @SMTP.EVENT.

@SOM

This system variable determines the character used to delineate the Start of Message. It defaults to ASCII SOH (01). The Start of Message detection can be disabled by setting @SOM to zero, in which case the message accumulates until @EOM is detected only.

@EOM

This system variable determines the character used to delineate the End of Message. It defaults to ASCII ETX (03).

@CONFIG

These system variables are used to access the ColorLCD configuration settings.

@CONFIG.ITEMS

This read-only system variable returns the total number of configuration setting items.

@CONFIG.TYPE[n]

This read-only system variable gets the type of the configuration item n:

@CONFIG.TYPE[n]	Item Type	Fields
1	Byte	0
2	Boolean	0
3	Unsigned short	0
4	Baudrate selector	0
5	Parity selector	0
6	Data Bits selector	0
7	Stop Bits selector	0
8	Keybeep selector	0
9	Firmware Version	0
10	Keypad style	0
11	Keypad scheme	0
12	Protocol selector	0
13	MAC address	6
14	IP address (only display if static)	4
15	IP address	4
16	Hex Byte	0
17	Hex Unsigned short	0
18	Hex Array	8
19	Short	0
20	RS485 Mode	0

@CONFIG.NAME$[n]

This read-only system variable gets the name of the configuration item n.

@CONFIG.VALUE$[n{, f}]

This read-only system variable gets the human readable value of the configuration item n. If the item has fields then the second argument specifies the zero-based field number.

@CONFIG.MIN[n]

This read-only system variable gets the allowed minimum value of the configuration item n.

@CONFIG.MAX[n]

This read-only system variable gets the allowed maximum value of the configuration item n.

@CONFIG.FIELDS[n]

This read-only system variable gets the number of fields of the configuration item n.

@CONFIG.FIELD$[n, f]

This read-only system variable gets the human readable value of the configuration item n's field f.

@CONFIG.SEPARATOR$[n, f]

This read-only system variable gets the human readable value of the configuration item n's field f separator.

@CONFIG.VALUE[n{, f}]

This system variable gets or sets the value of the configuration item n. If the item has fields then the second argument specifies the zero-based field number.

@CONFIG.DEFAULT[n{, f}]

This read-only system variable gets the default value of the configuration item n. If the item has fields then the second argument specifies the zero-based field number.

@CONFIG.WRITE[n, f]

Changes to the configuration settings affect the current value of RAM copies of the entries. Setting this write-only system variable to one forces the current value of item n to be written to the NVM backup store so that it will be remembered between restarts. If the item has fields then the second argument specifies the zero-based field number.

@CARD.MOUNT

This system variable is used to mount and un-mount the micro SD card.

Display System Variables

ACS Basic also has built-in display system variables. System variables are denoted by a '**@**' character as the first character of the variable name. The system variable names are 'tokenized' when entered to save program memory and speed program execution: for example the system variable **@SECOND** would be tokenized to two bytes instead of seven bytes.

System variables **may not** be assigned a value by appearing in an **FOR, DIM, INPUT, READ, FINPUT #N** or **FREAD #N** statement.

Some system variables are read-only and may not appear on the left hand side of a **LET** assignment statement.

Some system variables have ***Events*** associated with them and may be referenced in **ONEVENT, SIGNAL** and **WAIT** statements. See the description for the individual system variables and the ***Events*** section below for more information.

@ANSI.

These system variables are used to how and when Basic PRINT statements also affect the display.

@ANSI.ENABLE

This system variable enables or disables whether PRINT statements are also sent to the display as ANSI text. The default is enabled (1). See the PRINT statement below for additional information.

@ANSI.CURSOR

This system variable enables or disables whether a blinking cursor is displayed on the display when @ANSI.ENABLE=1. The default is enabled (1).

@ANSI.SCROLL

his system variable enables or disables whether a line feed on the last line of the display causes the screen to scroll up when @ANSI.ENABLE=1. The default is enabled (1).

@ANSI.WRAP

This system variable enables or disables whether PRINTed text on the display wraps to the next line when it reaches the end of a line when @ANSI.ENABLE=1. The default is enabled (1).

@ANSI.ROW

This system variable gets or sets the current row where PRINTed text will be placed.

@ANSI.COL

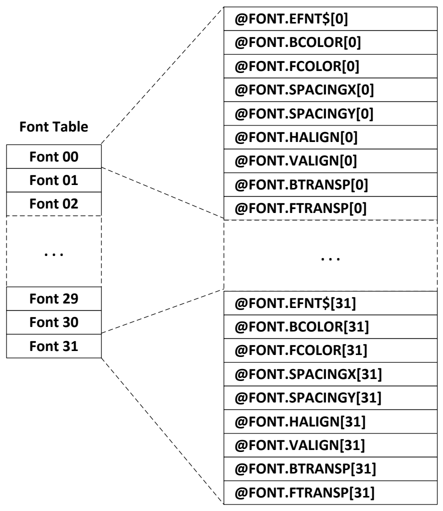
This system variable gets or sets the current column where PRINTed text will be placed.

@BACKLIGHT

This system variable controls the display backlight. Setting it to zero dims the backlight to the configured off setting, setting it to one brightens the backlight to the configured on setting and setting it to 2 brightens the backlight and starts the configured automatic backlight dim timer.

@FONT.

These system variables are used to access the individual elements of the Font table. They require an index that ranges from 0 to 31 to refer to the 32 Font table entries.



@FONT.EFNT$[n]

This system variable sets or gets the embedded font resource name in the resource table for this Font table entry.

@FONT.BCOLOR[n]

This system variable sets or gets the background color for Font table entry n:

@FONT.BCOLOR[n] = RGB(R, G, B)

Where:

Field	Description
`n`	Font table entry #
`RGB(R, G, B)`	Background color as RGB565

@FONT.FCOLOR[n]

This system variable sets the foreground color for Font table entry n:

@FONT.FCOLOR[n] = RGB(R, G, B)

Where:

Field	Description
`n`	Font table entry #
`RGB(R, G, B)`	Background color as RGB565

@FONT.SPACINGX[n]

This system variable sets the horizontal spacing for Font table entry n:

@FONT.SPACINGX[n] = x

Where:

Field	Description
`n`	Font table entry #
`x`	0 = fixed width of the font 1→14 = pixels between characters 15 = bounding box character width

@FONT.SPACINGY[n]

This system variable sets the vertical spacing for Font table entry n:

@FONT.SPACINGY[n] = y

Where:

Field	Description
`n`	Font table entry #
`y`	0 = fixed width of the font 1→14 = pixels between characters 15 = bounding box character width

@FONT.HALIGN[n]

This system variable sets the horizontal alignment (justification) for Font table entry n:

@FONT.HALIGN[n] = h

Where:

Field	Description
`n`	Font table entry #
`h`	0 = Left Justify 1 = Center Justify 2 = Right Justify

@FONT.VALIGN[n]

This system variable sets the vertical alignment (justification) for Font table entry n:

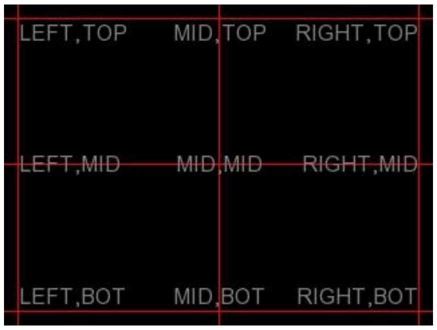
@FONT.VALIGN[n] = v

Where:

Field	Description
`n`	Font table entry #
`v`	0 = Top Justify 1 = Center Justify 2 = Bottom Justify

This short program shows the effect of the @FONT.HALIGN and @FONT.VALIGN system variables:

```
5 REM Test @FONT.HALIGN and @FONT.VALIGN
10 @ANSI.CURSOR=0:@ANSI.ENABLE=0:@BACKLIGHT=1
20 REM Draw grid
30 DRAW.LINE 10,0,10,239,RGB(255,0,0)
40 DRAW.LINE 160,0,160,239,RGB(255,0,0)
50 DRAW.LINE 309,0,309,239,RGB(255,0,0)
60 DRAW.LINE 0,10,319,10,RGB(255,0,0)
70 DRAW.LINE 0,120,319,120,RGB(255,0,0)
80 DRAW.LINE 0,229,319,229,RGB(255,0,0)
90 REM Draw aligned text
100 @FONT.HALIGN[0]=0:@FONT.VALIGN[0]=0:DRAW.TEXT 0,10,10,"LEFT,BOT"
110 @FONT.HALIGN[0]=1:@FONT.VALIGN[0]=0:DRAW.TEXT 0,160,10,"MID,BOT"
120 @FONT.HALIGN[0]=2:@FONT.VALIGN[0]=0:DRAW.TEXT 0,309,10,"RIGHT,BOT"
130 @FONT.HALIGN[0]=0:@FONT.VALIGN[0]=1:DRAW.TEXT 0,10,120,"LEFT,MID"
140 @FONT.HALIGN[0]=1:@FONT.VALIGN[0]=1:DRAW.TEXT 0,160,120,"MID,MID"
150 @FONT.HALIGN[0]=2:@FONT.VALIGN[0]=1:DRAW.TEXT 0,309,120,"RIGHT,MID"
160 @FONT.HALIGN[0]=0:@FONT.VALIGN[0]=2:DRAW.TEXT 0,10,229,"LEFT,TOP"
170 @FONT.HALIGN[0]=1:@FONT.VALIGN[0]=2:DRAW.TEXT 0,160,229,"MID,TOP"
180 @FONT.HALIGN[0]=2:@FONT.VALIGN[0]=2:DRAW.TEXT 0,309,229,"RIGHT,TOP"
190 GOTO 190
```



@FONT.BTRANSP[n]

This system variable sets the background transparency for Font table entry n:

@FONT.BTRANSP[n] = b

Where:

Field	Description
`n`	Font table entry #
`b`	0 = Solid 1 = Transparent

@FONT.FTRANSP[n]

This system variable sets the foreground transparency for Font table entry n:

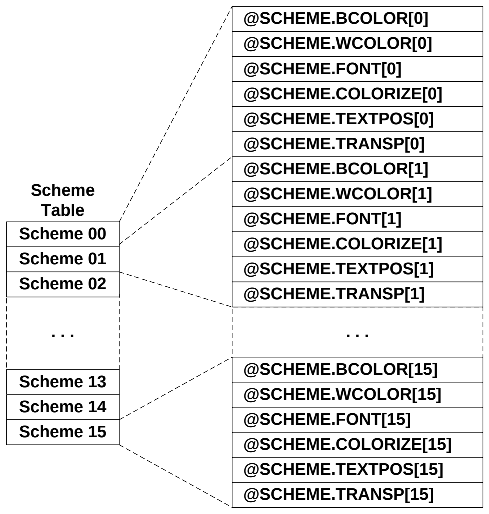
@FONT.FTRANSP[n] = b

Where:

Field	Description
`n`	Font table entry #
`b`	0 = Solid 1 = Transparent

@SCHEME.

These system variables are used to access the individual elements of the Scheme table. They require an index that ranges from 0 to 15 to refer to the 16 scheme table entries. Schemes are used in pairs with the lower numbered scheme entry used for rendering 'un-pressed' or inactive screen icons and the higher number entry of the pair used for rendering 'pressed' or active screen icons.



@SCHEME.FONT[n]

This system variable gets or sets the font table entry number to be used for Scheme table entry n:

@SCHEME.FONT[n] = f

Where:

Field	Description
`n`	Scheme table entry #
`f`	0-31 = Fonts table entry #

@SCHEME.BCOLOR[n]

This system variable gets or sets the black replace color to be used for Scheme table entry:

@SCHEME.BCOLOR[n] = RGB(R, G, B)

Where:

Field	Description
`n`	Scheme table entry #
`RGB(R, G, B)`	Black replacement color as RGB565

@SCHEME.WCOLOR[n]

This system variable gets or sets the white replace color to be used for Scheme table entry n:

@SCHEME.WCOLOR[n] = RGB(R, G, B)

Where:

Field	Description
`n`	Scheme table entry #
`RGB(R, G, B)`	White replacement color as RGB565

@SCHEME.COLORIZE[n]

This system variable gets or sets the colorization mode to be used for Scheme table entry n:

@SCHEME.COLORIZE[n] = c

Where:

Field	Description
`n`	Scheme table entry #
`c`	0 = no change 1 = grayscale 2 = shade

@SCHEME.TEXTPOS[n]

This system variable gets or sets the text position to be used for this Scheme table entry n:

@SCHEME.TEXTPOS[n] = p

Where:

Field	Description
`n`	Scheme table entry #
`p`	0 = no text 1 = centered 2-511 = X, Y position

@SCHEME.TRANSP[n]

This system variable gets or sets the transparency mode to be used for Scheme table entry n:

@SCHEME.TRANSP[n] = m

Where:

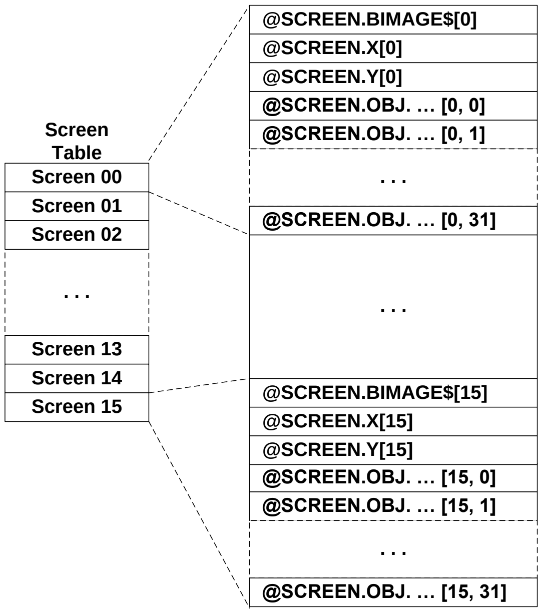
Field	Description
`n`	Schemes table entry #
`m`	(see following table)

Where:

m	Operation	Transparency Color Value	Description
0	Magic Value	Magic index	For indexed .BMP images only, the last index value is the magic color.
1	None	Not used	Surface pixel = image pixel.
2	Darken	Scheme white replace color to be used as a multiplier	Surface pixel = surface pixel * RGB565 color.
3	Source Color Filter	Not used	Surface pixel = surface pixel * image pixel (5:6:5 channel by channel multiply).
4	Source Alpha Mask	Scheme white replace color to be used as a 'white' replacement	Surface pixel = (surface pixel * RG565 color) + (1 – image pixel) * surface pixel (treats image as an alpha mask, white solid, black transparent, intermediate blended).
5	Fill	Scheme white replace fill color	Surface pixel = RGB565 color
6	Transparency Blend	Scheme white replace color to use as an Alpha multiplier	Surface pixel = (RGB565 color * image pixel) + (1 – RGB565 color) * surface pixel (blends image into surface by ratio determined by RGB565 color).
7	Shadow	Scheme white replace color to use as an Alpha multiplier	Surface pixel = (RGB565 color * image pixel * surface pixel) + (1 – image pixel) * surface pixel (image is mask to apply multiplication of RGB565 color).

@SCREEN.

These system variables are used to access the individual elements of the Screen table. They require an index that ranges from 0 to 15 to refer to the 16 screen table entries.



@SCREEN.BIMAGE$[n]

This system variable is used to set the optional background image for the Screen table entry n to an image resource:

@SCREEN.BIMAGE$[n] = ”MyImageResource.bmp”

@SCREEN.X[n]

This system variable is used to set the lower left horizontal position of the displayed screen.

@SCREEN.Y[n]

This system variable is used to set the lower left vertical position of the displayed screen.

@SCREEN.#

This read-only system variable reflects the screen number of the last screen event.

@SCREEN.EVENT

This system variable reflects the last screen event. There are two screen events:

@SCREEN.EVENT	Event
1	Screen[@SCREEN.#] hidden by: SCREENS.CHANGETO or SCREENS.PUSHTO

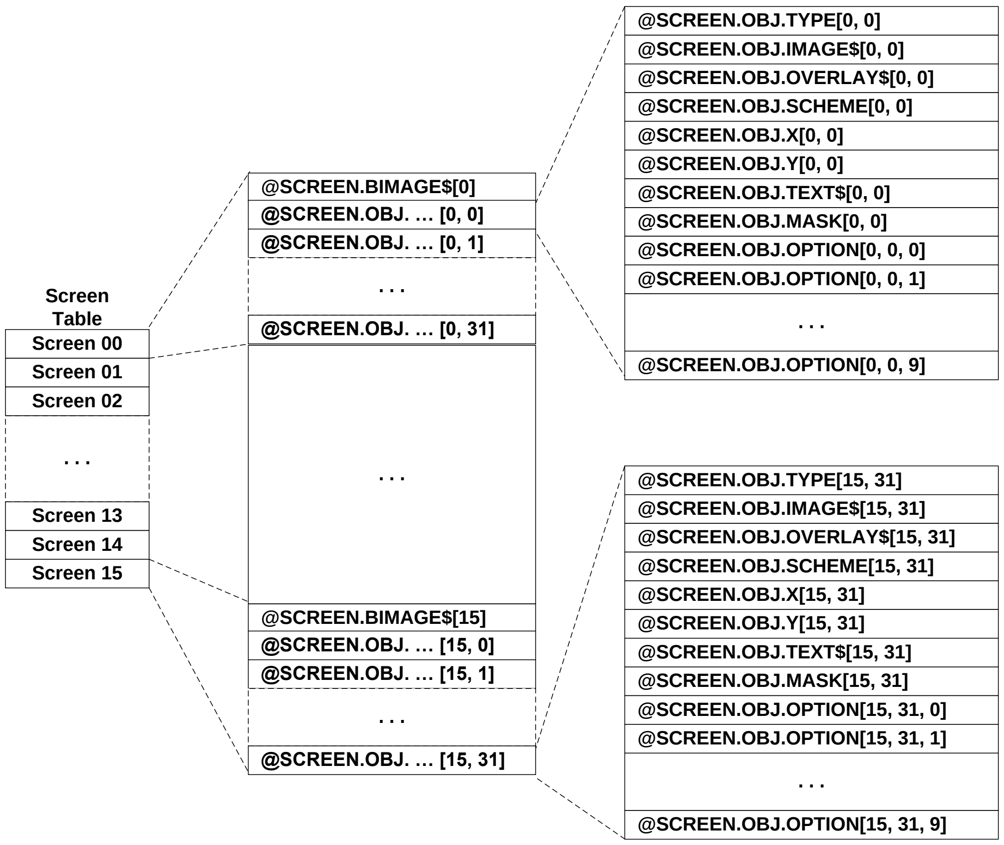
2	Screen[@SCREEN.#] shown by: SCREENS.CHANGETO or SCREENS.POP

@SCREEN.OBJ.

These system variables are used to access the individual attributes of the screen objects associated with each Screen table entry.

They require a double index – the first index selects the Screen table entry, and the second index selects the screen object entry for the selected screen entry. The first index ranges from 0 to 15 to refer to the 16 Screen table entries, and the second index ranges from 0 to 15 to refer to the 16 screen object table entries for each screen.

Screen object table entries have to be entered starting at zero and cannot have any gaps – unused entries will not be skipped over and processing stops at the first unused entry.



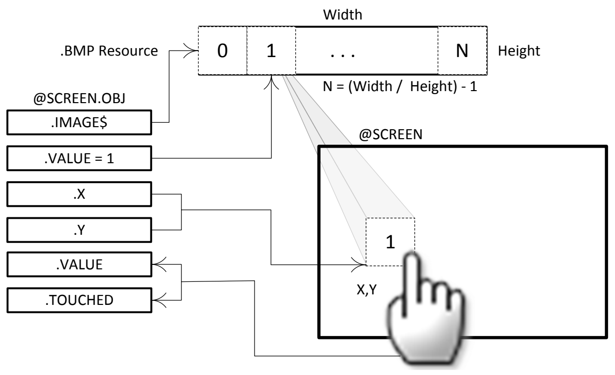
@SCREEN.OBJ.TYPE[screen, object]

This system variable sets the screen object type for Screens table entry screen, object entry:

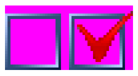
@SCREEN.OBJ.TYPE	Object Type
0	None
1	Icon
2	Button
3	Toggle Button
4	Back Button
5	Slider
6	Label
7	Touch Keypad
8	Radial Gauge
9	Linear Gauge
10	Listbox
11	Spinner Knob
12	Text Box

Icon Screen Object

The Icon screen object displays a portion of a bitmap resource where the displayed portion of the bitmap is controlled by the screen object's .VALUE. The displayed portion of the bitmap may be a square based upon the image width divided by the image height or a portion selected from the bitmap width divided by an .OPTION value. The auto-advance of the image portion when touched can be disabled by an .OPTION property.



A checkbox control can be implemented by using a .IMAGE$ bitmap resource containing the checkbox with and without the checkmark. Notice the magenta background which will become transparent when the control is rendered:



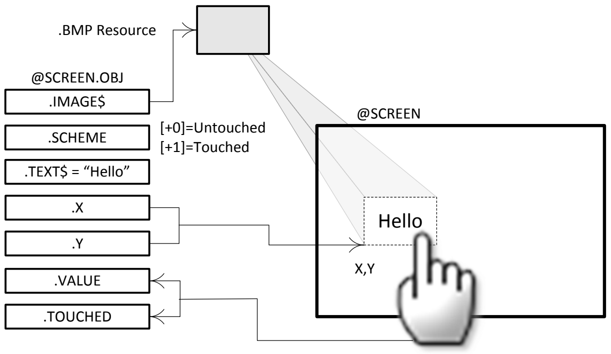
Here's a short program that puts this checkbox on the screen. When it is run the checkbox can be toggled between unchecked and checked by touching it:

```
5 REM Icon control used as Checkbox
10 @ANSI.ENABLE=0
20 @SCREEN.OBJ.TYPE[0,0]=1
30 @SCREEN.OBJ.IMAGE$[0,0]="Icon_CheckBox_Indexed_48x24.bmp"
40 @SCREEN.OBJ.X[0,0]=160:@SCREEN.OBJ.Y[0,0]=120
100 SCREENS.CHANGETO 0
999 GOTO 999
```

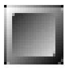
An animation capability can be achieved by advancing the object's .VALUE property on a timer event causing the icon screen object to advance through the contained images.

Button Screen Object

The Button screen object displays a bitmap with a .TEXT$ label that acts like an on-screen button. The button can auto-colorize for press/release feedback using the object's .SCHEME and can provide .EVENT notifications.



To support the pop-up keypad, the Color LCD has four built-in bitmap resources that it uses as buttons. These buttons are gray in color and are colorized by the screen object's .SCHEME when rendered:


ButtonK.bmp
22 x 22

ButtonK2.bmp
44 x 22

ButtonK4.bmp
88 x 22

ButtonM.bmp
48 x 32

Here's a simple program that displays one of those buttons – when it is run the button is colorized and it highlights when it is pressed on the screen:

```
10 REM Button Demo
15 @ANSI.ENABLE=0 : REM Disable ANSI text
20 @SCREEN.OBJ.TYPE[0,0]=2 : REM Set screen 0 object 0 to Button
25 @SCREEN.OBJ.SCHEME[0,0]=0 : REM Set button's scheme pair to 0,1
30 @SCREEN.OBJ.X[0,0]=116 : @SCREEN.OBJ.Y[0,0]=109 : REM Set button's location on screen
35 @SCREEN.OBJ.IMAGE$[0,0]="ButtonK4.bmp" : REM Set button's image
40 @SCREEN.OBJ.TEXT$[0,0]="Press" : REM Set button's text
45 SCREENS.CHANGETO 0 : REM activate screen 0
999 GOTO 999
```

Toggle Button Screen Object

The Toggle Button screen object displays a bitmap with the object's .TEXT$ label that acts like an on-screen latching button – each press/release toggles the button state. The button can auto-colorize for press/release feedback using the object's .SCHEME and provide .EVENT notifications. The same program can demonstrate the toggle functionality by simply changing the object type:

```
10 REM Toggle Button Demo
15 @ANSI.ENABLE=0 : REM Disable ANSI text
20 @SCREEN.OBJ.TYPE[0,0]=3 : REM Set screen 0 object 0 to Toggle Button
25 @SCREEN.OBJ.SCHEME[0,0]=0 : REM Set button's scheme pair to 0,1
30 @SCREEN.OBJ.X[0,0]=116 : @SCREEN.OBJ.Y[0,0]=109 : REM Set button's location on screen
35 @SCREEN.OBJ.IMAGE$[0,0]="ButtonK4.bmp" : REM Set button's image
40 @SCREEN.OBJ.TEXT$[0,0]="Press" : REM Set button's text
45 SCREENS.CHANGETO 0 : REM activate screen 0
999 GOTO 999
```

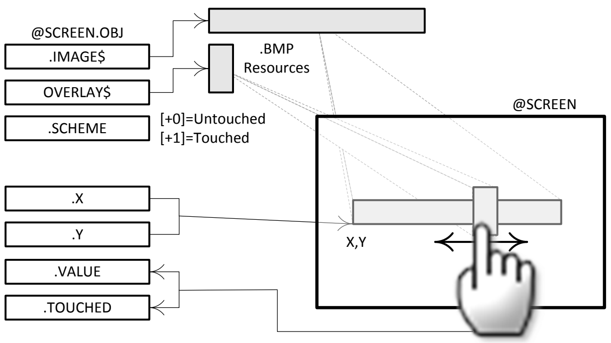
Back Button Screen Object

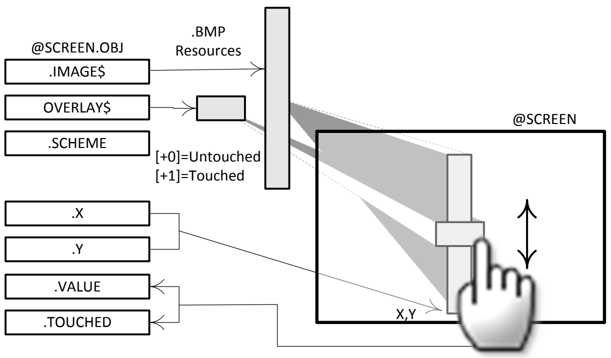
The Back Button screen object displays a bitmap with the object's .TEXT$ label that will pop the screen stack when pressed. The button can auto-colorize for press/release feedback using the object's .SCHEME and provide .EVENT notifications.

Slider Screen Object

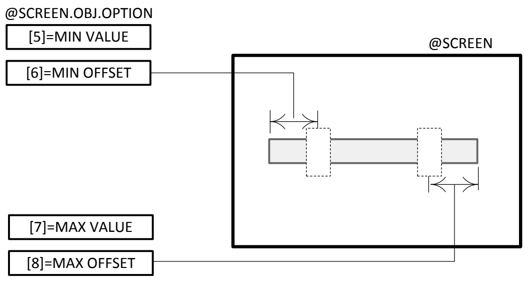
The Slider screen object displays a horizontal or vertical slider. The slider uses two bitmap resources configured with the .IMAGE$ and .OVERLAY$. The IMAGE$ bitmap resource supplies the slider base and the .OVERLAY$ bitmap resource supplies the slider button.

The slider orientation is controlled by the aspect ratio of the .IMAGE$ resource – bitmaps that are wider than they are tall operate horizontally and bitmaps that are taller than they are wide operate vertically.





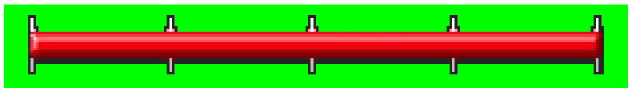
The slider screen object can be configured for min/max values and corresponding button center to base bitmap pixel offsets via the .OPTION properties.



If an .OPTION offset property is zero the corresponding offset is equal to the width (height) of the button bitmap.

If both of the .OPTION min and max value properties are zero the slider object's .VALUE ranges from 0 ↔ 32767.

Here are two bitmaps for a horizontal slider – the base .IMAGE$ and the button .OVERLAY$ Notice the green background that will become transparent when the control is rendered:

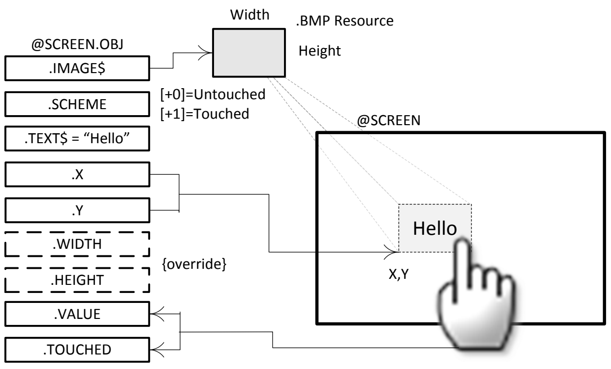


And here's a short program to initialize and display the slider:

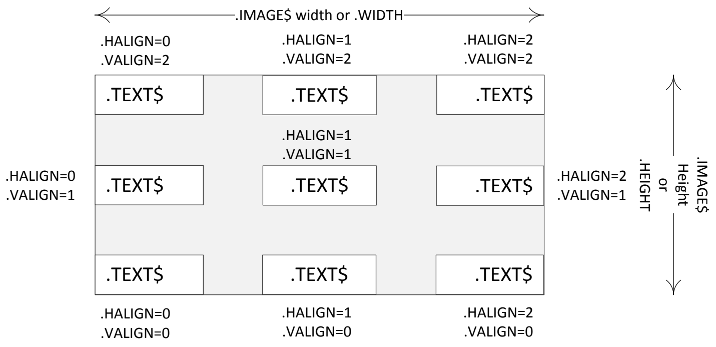
```
5 REM slider demo
10 @ANSI.ENABLE=0
15 @SCREEN.OBJ.TYPE[0,0]=5 : REM screen 0 object 0 slider
20 @SCREEN.OBJ.IMAGE$[0,0]="Slider_Red_H_WDivisions234x31.bmp"
25 @SCREEN.OBJ.OVERLAY$[0,0]="SliderButton_Red_H_19x26.bmp"
30 @SCHEME.COLORIZE[0]=0 : @SCHEME.TRANSP[0]=0 : @SCHEME.COLORIZE[1]=0 : @SCHEME.TRANSP[1]=0
35 @SCREEN.OBJ.SCHEME[0,0]=0
40 @SCREEN.OBJ.X[0,0]=(320-234)/2 : @SCREEN.OBJ.Y[0,0]=(240-31)/2
45 SCREENS.CHANGETO 0
999 GOTO 999
```

Label Screen Object

The Label screen object displays the object's .TEXT$ justified within the control's .IMAGE$ bitmap resource. The width and height of the displayed text area is controlled by the .IMAGE$ width and height or it can be overridden via the .WIDTH and .HEIGHT values. If no .IMAGE$ bitmap resource is specified a box of the configured TPAD_Scheme's off color is drawn if the object's .SCHEME's .FONT's background is not transparent.



The position of the .TEXT$ within the label boundary is controlled by the .SCHEME font justification – the object's scheme FONT.HALIGN and FONT.VALIGN properties:

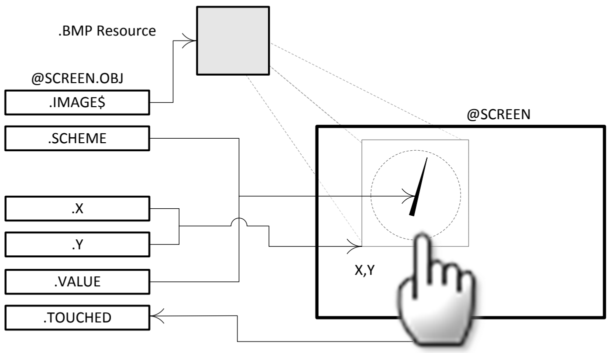


Touch Keypad Screen Object

The Touch Keypad screen object provides the ability for the screen to pop-up and display several different on-screen keypads and receive the pressed characters as input.

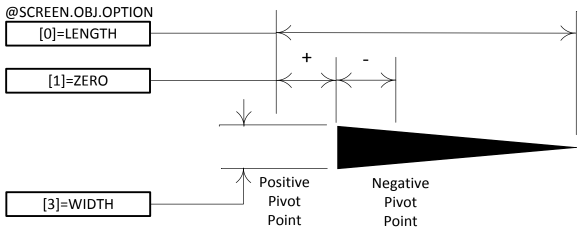
Radial Gauge Screen Object

The Radial Gauge screen object displays the .IMAGE$ bitmap resource with an overlaid graphical needle that can be justified to the bitmap and configured for angle and value range using .OPTION properties. The object's .VALUE then sets the needle angle.

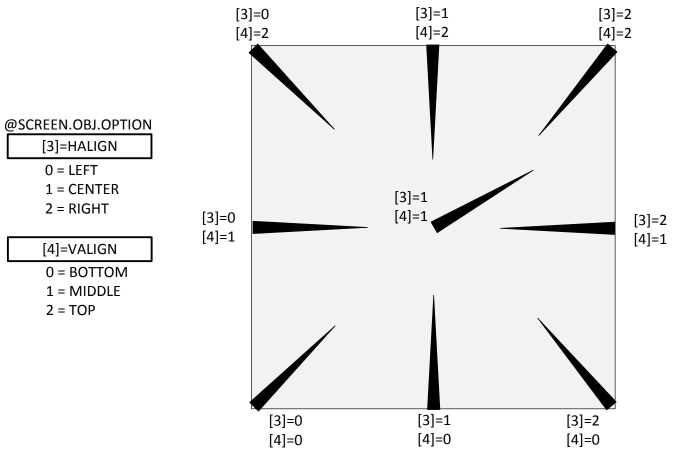


The first three .OPTION properties set the gauge needle length, pivot point offset and width in pixels. A positive pivot point offset allows the needle to rotate about a point before its base. A negative pivot point offset allows the needle to rotate about a point along its length.

Shorter, skinnier needles draw faster.



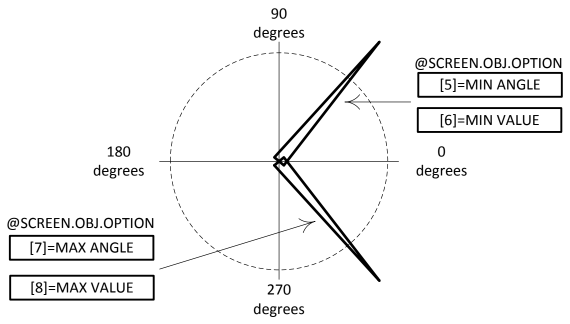
The next two .OPTION properties align the needle's pivot point to the .IMAGE$ bitmap resource.



The last four .OPTION properties configure the needle's minimum and maximum angles and the corresponding minimum and maximum values. Both the angles and values can be negative. Angles advance in value counter-clockwise rotation with zero to the right as shown in the following diagram.

The minimum angle is the angle the needle will be drawn at when the object's .VALUE is at the specified minimum value.

The maximum angle is the angle the needle will be drawn at when the object's .VALUE is at the specified maximum value.



Here is a bitmap for a tachometer radial gauge:

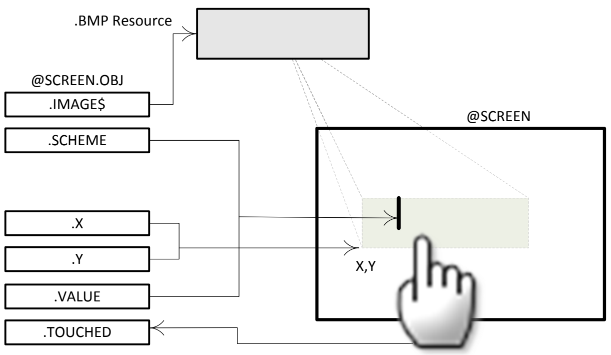
0
1
2
3
4
5
6
7
8
9
x 1000 r/min

And here's a short program that initializes a screen object as a radial gauge using this bitmap, then runs the tach from 0 to 10,000 and back down in a loop:

```
5 REM Tachometer
10 @ANSI.ENABLE=0:@BACKLIGHT=1
15 @SCHEME.WCOLOR[0]=RGB(255,0,0):REM needle color
20 @SCREEN.OBJ.TYPE[0,0]=8:REM radial gauge
25 @SCREEN.OBJ.OPTION[0,0,0]=95:REM needle length
30 @SCREEN.OBJ.OPTION[0,0,1]=18:REM needle zero
35 @SCREEN.OBJ.OPTION[0,0,2]=6:REM needle width
40 @SCREEN.OBJ.OPTION[0,0,3]=1:REM needle halign
45 @SCREEN.OBJ.OPTION[0,0,4]=1:REM needle valign
50 @SCREEN.OBJ.OPTION[0,0,5]=0:REM min value
55 @SCREEN.OBJ.OPTION[0,0,6]=225:REM min angle
60 @SCREEN.OBJ.OPTION[0,0,7]=10000:REM max value
65 @SCREEN.OBJ.OPTION[0,0,8]=-25:REM max angle
70 @SCREEN.OBJ.IMAGE$[0,0]="Gauge-Tach-Black-3QTR-196x196.bmp"
75 SCREENS.CHANGETO 0
80 WAIT @SCREEN.EVENT
95 FOR n=0 TO 10000 STEP 20:@SCREEN.OBJ.VALUE[0,0]=n:NEXT n
100 FOR n=10000 TO 0 STEP -20:@SCREEN.OBJ.VALUE[0,0]=n:NEXT n
105 GOTO 95
```

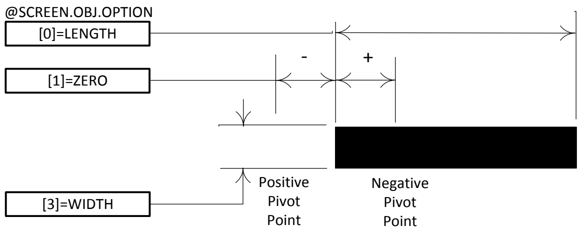
Linear Gauge Object

The Linear Gauge screen object display the .IMAGE$ bitmap resource with overlaid graphical needle that can be justified to the bitmap and configured for offset and value range using the .OPTION properties. The object's .VALUE then sets the needle position.

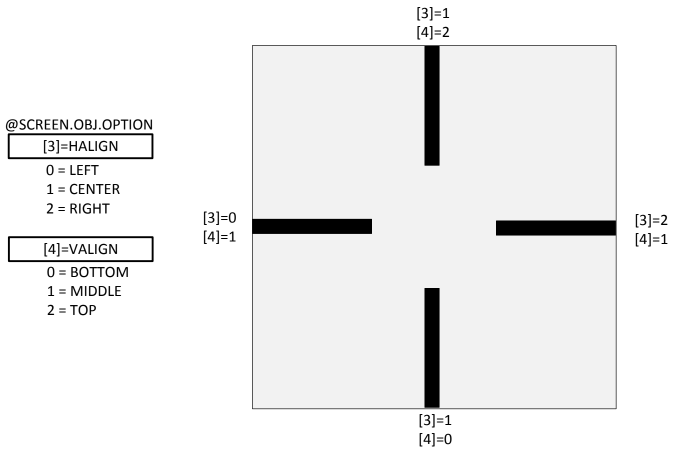


The first three .OPTION properties set the gauge needle length, zero offset and width in pixels. A positive zero offset allows the needle to move along a line parallel to and inside its base. A negative offset allows the needle to move along a line parallel to and outside of its base.

Shorter, skinnier needles draw faster.



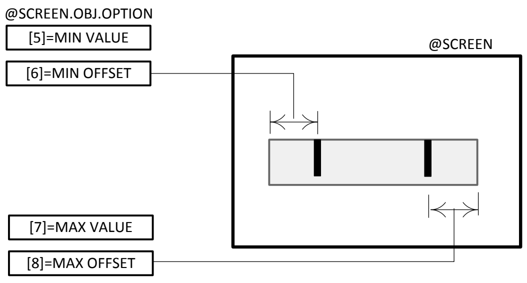
The next two .OPTION properties align the needle to the .IMAGE$ bitmap resource.



The last four .OPTION properties configure the needle's minimum and maximum offsets and the corresponding minimum and maximum values. Both the offsets and values can be negative.

The minimum offset is the offset from the .IMAGE$ edge the needle will be drawn at when the object's .VALUE is at the specified minimum value.

The maximum offset is the offset from the .IMAGE$ edge the needle will be drawn at when the object's .VALUE is at the specified maximum value.



Listbox Screen Object

The Listbox screen object displays the .IMAGE$ bitmap resource with overlaid up/down buttons on the right hand side and the object's DATA displayed as text strings using the object's .SCHEME font.

If no .IMAGE$ bitmap resource is specified a box of the configured TPAD_Scheme's off color is drawn if the object's .SCHEME's .FONT's background is not transparent. A bounding box of the .WIDTH and .HEIGHT is then drawn in the configured TPAD_Scheme's on color.

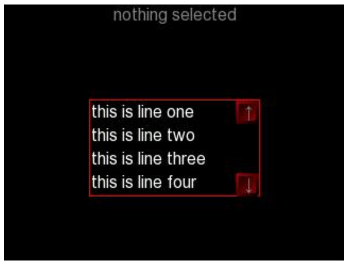
The number of displayed items and the topmost item in the list may be controlled by .OPTION properties.

An item may be selected or deselected by touching. The object's .VALUE reflects the selected item number with minus one (-1) indicating no item is selected.

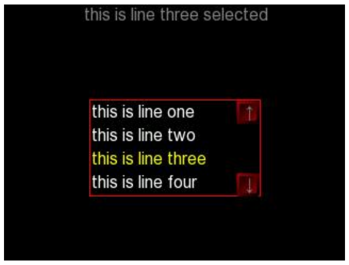
Here's a short program that displays a listbox control with some character strings that displays the selected item at the top of the screen in a label control:

```
10 REM listbox
15 @ANSI.ENABLE = 0 : @BACKLIGHT = 1
20 INCLUDE "constants.bas"
25 CONST LBox = 0 : @SCREEN.OBJ.TYPE[0,LBox] = Listbox
30 REM @SCREEN.OBJ.IMAGE$[0,0] = "PopUp_160x120.bmp"
35 @FONT.HALIGN[8] = 0 : @FONT.HALIGN[9] = 0 : @SCREEN.OBJ.SCHEME[0,0] = 8
40 @FONT.FCOLOR[9] = RGB(255,255,0)
45 @SCREEN.OBJ.X[0,LBox]=ScreenWidth / 2 - 160 / 2 : @SCREEN.OBJ.Y[0,LBox]=ScreenHeight / 2 - 120 / 2
50 @SCREEN.OBJ.WIDTH[0,LBox] = 160 : @SCREEN.OBJ.HEIGHT[0,LBox] = 90
55 CONST Readout=1 : @SCREEN.OBJ.TYPE[0,Readout] = Label
60 @SCREEN.OBJ.X[0,Readout] = 160-100 : @SCREEN.OBJ.Y[0,Readout] = 220
65 @SCREEN.OBJ.WIDTH[0,Readout] = 200 : @SCREEN.OBJ.HEIGHT[0,Readout] = 20
70 @SCREEN.OBJ.MASK[0,Readout] = None
75 DIM items$[9]
80 items$[0] = "this is line one" : items$[1] = "this is line two" : items$[2] = "this is line three"
85 items$[3] = "this is line four" : items$[4] = "this is line five" : items$[5] = "this is line six"
90 items$[6] = "this is line seven" : items$[7] = "this is line eight" : items$[8] = "this is line nine"
95 SCREEN.OBJ.DATA 0,LBox,0,items$ : @SCREEN.OBJ.OPTION[0,LBox,NumberOfItems] = UBOUND(items$)
100 ONEVENT @SCREEN.OBJ.EVENT,GOSUB `Handler
105 SCREENS.CHANGETO 0
110 WAIT @SCREEN.EVENT
115 @SCREEN.OBJ.VALUE[0,LBox] = -1 : REM nothing selected
999 GOTO 999
1000 `Handler : REM screen object event handler
1005 LIF @SCREEN.OBJ.# = LBox THEN ON @SCREEN.OBJ.EVENT-1,GOTO 0,`Selector,0
1010 RETURN
1015 `Selector : IF @SCREEN.OBJ.VALUE[0,LBox] >= 0 THEN
1020 @SCREEN.OBJ.TEXT$[0,Readout] = items$[@SCREEN.OBJ.VALUE[0,LBox]] + " selected"
1025 ELSE
1030 @SCREEN.OBJ.TEXT$[0,Readout] = "nothing selected"
1035 ENDIF
1040 RETURN
```

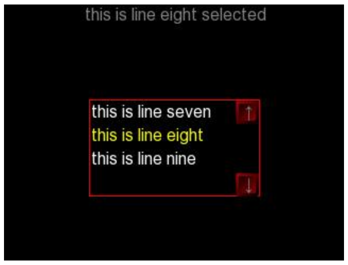
And here is the resulting display:


nothing selected

this is line one
this is line two
this is line three
this is line four


this is line three selected

this is line one
this is line two
this is line three
this is line four

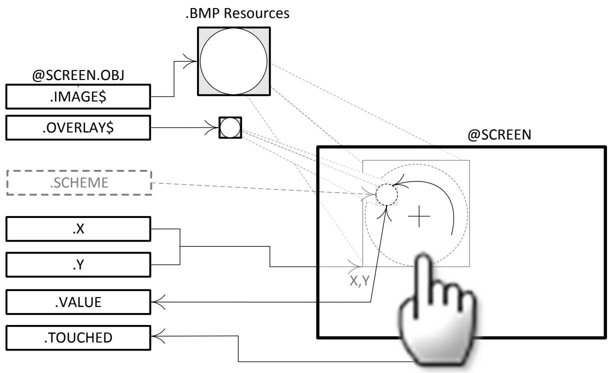

this is line eight selected

this is line seven
this is line eight
this is line nine

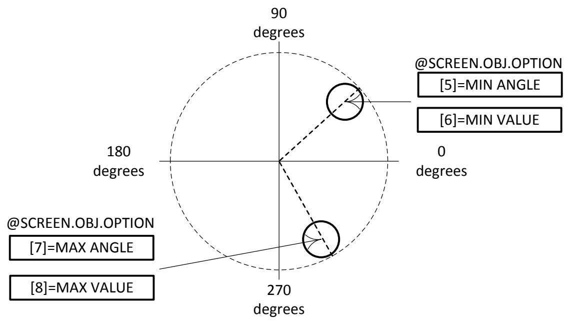
Spinner Knob Object

The Spinner Knob screen object displays a rotary knob with a smaller spinner handle or radial indicator line. The slider uses two bitmap resources configured with the .IMAGE$ and .OVERLAY$. The IMAGE$ bitmap resource supplies the base rotary know and the .OVERLAY$ bitmap resource supplies the spinner handle. The object's .VALUE gets (or sets) the position of the .OVERLAY$ bitmap resource in an arc relative to the .IMAGE$ bitmap resource's center.

If an .OVERLAY$ image is not provided, a radial indicator line is drawn instead. The color of the radial line is controlled by the screen object's .SCHEME.WCOLOR[0], and the indicator length, zero and width are controlled by three screen object .OPTION values.



The spinner handle / indicator position and value relationship is controlled by four screen object .OPTION values.



Here are two bitmaps for a spinner knob screen object – the base .IMAGE$ and the spin handle .OVERLAY$. Notice the magenta background that will become transparent when the control is rendered:

Spinner_33x33.bmp

Knob_150x150.bmp

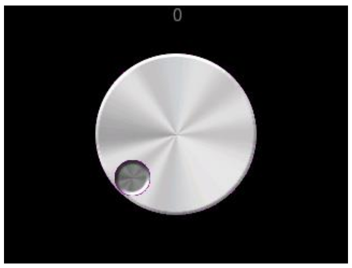
And here is a sample program that displays the knob at the center of the screen along with a label at the top of the screen showing the .VALUE as the knob is rotated by touching and turning:

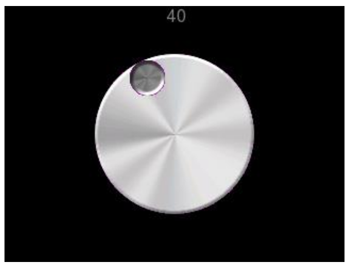
```
10  REM spinner knob
15  INCLUDE "constants.bas"
20  @ANSI.ENABLE = False : @BACKLIGHT = True
25  CONST Knob=0 : @SCREEN.OBJ.TYPE[0,Knob] = SpinnerKnob
30  @SCREEN.OBJ.IMAGE$[0,Knob]="Knob_150x150.bmp" : @SCREEN.OBJ.OVERLAY$[0,Knob]="Spinner_33x33.bmp"
35  @SCREEN.OBJ.X[0,Knob]=ScreenWidth / 2 - 150 / 2 : @SCREEN.OBJ.Y[0,Knob]=ScreenHeight / 2 - 150 / 2
40  @SCREEN.OBJ.OPTION[0,Knob,MinValue] = 0 : @SCREEN.OBJ.OPTION[0,Knob,MinAngle] = 225
45  @SCREEN.OBJ.OPTION[0,Knob,MaxValue] = 100 : @SCREEN.OBJ.OPTION[0,Knob,MaxAngle] = -45
50  CONST Readout=1 : @SCREEN.OBJ.TYPE[0,Readout] = Label
52  @SCREEN.OBJ.X[0,Readout] = 160-25 : @SCREEN.OBJ.Y[0,Readout] = 220
55  @SCREEN.OBJ.WIDTH[0,Readout] = 50 : @SCREEN.OBJ.HEIGHT[0,Readout] = 20
60  @SCREEN.OBJ.MASK[0,Knob] = ValueChanged : @SCREEN.OBJ.MASK[0,Readout] = None
65  ONEVENT @SCREEN.OBJ.EVENT, GOSUB 10000
100  SCREENS.CHANGETO 0
110  WAIT @SCREEN.EVENT
115  @SCREEN.OBJ.VALUE[0,Knob] = 0
120  GOTO 120
10000  REM screen object event handler
10005  @SCREEN.OBJ.TEXT$[0,Readout] = STR$(@SCREEN.OBJ.VALUE[0,Knob])
10010  RETURN
```

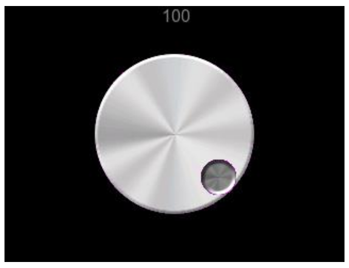
In order to make the program easier to follow it includes a file “constants.bas” that defines several constant variables:

```
CONST False=0,True=1
CONST ScreenWidth=320,ScreenHeight=240
REM Screen Object Types
CONST None=0,Icon=1,Button=2,ToggleButton=3,BackButton=4,Slider=5,Label=6,TouchKeypad=7,RadialGauge=8
CONST LinearGauge=9,Listbox=10,SpinnerKnob=11,Textbox=12
REM Alignment and Transparencies
CONST Left=0,Center=1,Right=2,Top=0,Middle=1,Bottom=2,Solid=0,Transparent=1
REM Drawing Surfaces
CONST Display=0,Work=1,Background=2
REM Screen Object Event Mask bits
CONST TouchEvents=1,ValueEvents=2,ReleasedEvents=4
REM Screen Object Events
CONST Touched=1,ValueChanged=2,Released=3
REM SpinnerKnob Options
CONST IndicatorLength=0,IndicatorZero=1,IndicatorWidth=2,MinValue=5,MinAngle=6,MaxValue=7,MaxAngle=8
REM + Slider Options
CONST MinOffset=6,MaxOffset=8
REM + Gauge and Icon Options
CONST NumberOfImages=0,Attributes=1,DisableAdvance=1,HAlign=3,VAlign=4
REM + Listbox Options
CONST NumberOfItems=0,TopItem=1,ItemHeight=2
REM + Textbox Options
CONST Style=0,NumberOfChars=1
REM + Textbox Styles
CONST NumbersOnly=1,Password=2,Lowercase=4,Uppercase=8,Readonly=16
```

And here is the resulting screen display:



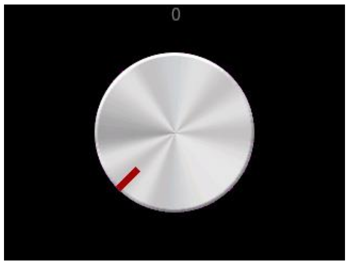


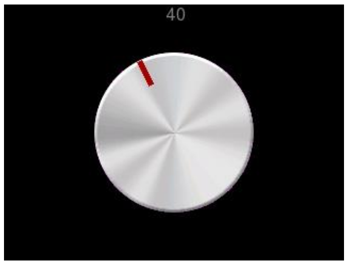


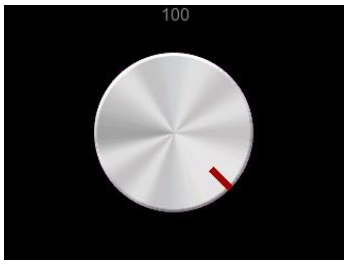
To use an indicator knob without the spinner don’t supply the spinner bitmap and initialize the three indicator options:

```
10  REM knob with indicator
15  INCLUDE "constants.bas"
20  @ANSI.ENABLE = False : @BACKLIGHT = True
25  CONST Knob=0 : @SCREEN.OBJ.TYPE[0,Knob] = SpinnerKnob
30  @SCREEN.OBJ.IMAGE$[0,Knob] = "Knob_150x150.bmp"
35  @SCREEN.OBJ.X[0,Knob]=ScreenWidth / 2 - 150 / 2 : @SCREEN.OBJ.Y[0,Knob]=ScreenHeight / 2 - 150 / 2
37  @SCREEN.OBJ.OPTION[0,Knob,IndicatorLength] = 75 : @SCREEN.OBJ.OPTION[0,Knob,IndicatorZero] = 50
38  @SCREEN.OBJ.OPTION[0,Knob,IndicatorWidth] = 5
40  @SCREEN.OBJ.OPTION[0,Knob,MinValue] = 0 : @SCREEN.OBJ.OPTION[0,Knob,MinAngle] = 225
45  @SCREEN.OBJ.OPTION[0,Knob,MaxValue] = 100 : @SCREEN.OBJ.OPTION[0,Knob,MaxAngle] = -45
50  CONST Readout=1 : @SCREEN.OBJ.TYPE[0,Readout]=Label
52  @SCREEN.OBJ.X[0,Readout] = 160-25 : @SCREEN.OBJ.Y[0,Readout] = 220
55  @SCREEN.OBJ.WIDTH[0,Readout] = 50 : @SCREEN.OBJ.HEIGHT[0,Readout] = 20
60  @SCREEN.OBJ.MASK[0,Knob] = ValueChanged : @SCREEN.OBJ.MASK[0,Readout] = None
65  ONEVENT @SCREEN.OBJ.EVENT, GOSUB 10000
100  SCREENS.CHANGETO 0
110  WAIT @SCREEN.EVENT
115  @SCREEN.OBJ.VALUE[0,Knob] = 0
120  GOTO 120
10000  REM screen object event handler
10005  @SCREEN.OBJ.TEXT$[0,Readout] = STR$(@SCREEN.OBJ.VALUE[0,Knob])
10010  RETURN
```

And the result is:







Textbox Object

The Textbox screen object displays the object's .TEXT$ justified within the control's .IMAGE$ bitmap resource.

The width and height of the displayed text area is controlled by the .IMAGE$ width and height or it can be overridden via the .WIDTH and .HEIGHT values. If no .IMAGE$ bitmap resource is specified a box of the configured TPAD_Scheme's off color is drawn outlined by a box of the configured TPAD_Scheme's on color if the object's .SCHEME's .FONT's background is not transparent.

The justification and color of the object's .TEXT$ value is controlled by the font specified object's .SCHEME pair – unfocused is drawn using the first scheme in the pair, focused is drawn using the second scheme in the pair.

If the Textbox is touched, the keypad pops up and the outline rectangle changes from solid to dashed and animates to show that the object has focus. The keypad is popped above or below the control depending upon the control's .Y coordinate so as not to obscure the it. Textboxes that are styled as Numbers Only present the numeric keypad.

The current contents of the .TEXT$ value is selected. Keys that are typed on the popup keypad are entered into the object's .TEXT$ value, replacing the selection. A backspace (Bs) key removes the last key pressed. A carriage return (Enter) key un-focuses the control and hides the pop-up keypad. Each typed key is reported as a change in the screen object's .VALUE along with a value changed event.

There are several textbox styles that are configurable as bit values in the object's .OPTION register. As these are binary weighted bit values they may be combined – for example a password textbox that only accepts numbers would be styled using a value of 1 + 2 = 3. A style of Uppercase only takes precedence over Lowercase only. A style of Read Only performs focusing and keypad pop-up, but the .TEXT$ value cannot be changed by key presses, only by code changes to the .TEXT$ value.

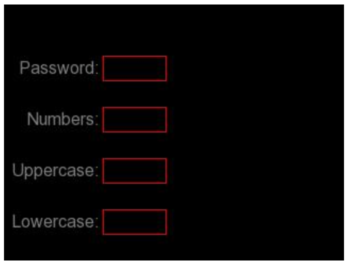
The available textbox style are:

Style bit	*function*
1	Numbers Only
2	Password (only '*' displayed)
4	Lowercase Only
8	Uppercase Only
16	Read Only

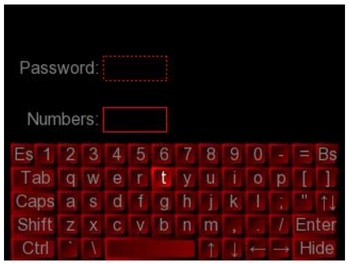
Here's a program that shows four textboxes with different styles and four label controls to receive the values entered into the textboxes when the Enter key is pressed.

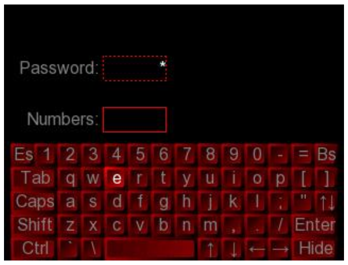
```
10 REM textboxes
15 @ANSI.ENABLE = 0 : @BACKLIGHT = 1
20 INCLUDE constants.bas
25 CONST EnterKey = 13
30 x = 0 : y = ScreenHeight - 72
35 FOR n = 0 TO 11 STEP 3
40   @SCREEN.OBJ.TYPE[0,(n+0)] = Label
45   @SCREEN.OBJ.X[0,(n+0)] = x : @SCREEN.OBJ.Y[0,(n+0)] = y - ((n/3)*48)
50   @SCREEN.OBJ.WIDTH[0,(n+0)] = 90 : @SCREEN.OBJ.HEIGHT[0,(n+0)] = 24
55   @SCREEN.OBJ.TYPE[0,(n+1)] = Textbox
60   @SCREEN.OBJ.X[0,(n+1)] = x + 93 : @SCREEN.OBJ.Y[0,(n+1)] = y - ((n/3)*48)
65   @SCREEN.OBJ.WIDTH[0,(n+1)] = 60 : @SCREEN.OBJ.HEIGHT[0,(n+1)] = 24
70   @SCREEN.OBJ.MASK[0,(n+1)] = ValueEvents
75   @SCREEN.OBJ.TYPE[0,(n+2)] = Label
80   @SCREEN.OBJ.X[0,(n+2)] = 160 : @SCREEN.OBJ.Y[0,(n+2)] = y - ((n/3)*48)
85   @SCREEN.OBJ.WIDTH[0,(n+2)] = 150 : @SCREEN.OBJ.HEIGHT[0,(n+2)] = 24
90 NEXT n
95 @FONT.HALIGN[0] = Right : @FONT.HALIGN[1] = Right
100 @SCREEN.OBJ.TEXT$[0,0] = "Password:" : @SCREEN.OBJ.OPTION[0,1,Style] = Password
105 @SCREEN.OBJ.TEXT$[0,3] = "Numbers:" : @SCREEN.OBJ.OPTION[0,4,Style] = NumbersOnly
110 @SCREEN.OBJ.OPTION[0,4,NumberOfChars] = 4
115 @SCREEN.OBJ.TEXT$[0,6] = "Uppercase:" : @SCREEN.OBJ.OPTION[0,7,Style] = Uppercase
120 @SCREEN.OBJ.TEXT$[0,9] = "Lowercase:" : @SCREEN.OBJ.OPTION[0,10,Style] = Lowercase
125 ONEVENT @SCREEN.OBJ.EVENT,GOSUB 1000
130 SCREENS.CHANGETO 0
135 WAIT @SCREEN.EVENT
999 GOTO 999
1000 REM screen object event handler
1005 ON @SCREEN.OBJ.EVENT, GOSUB 0,0,1015,0
1010 RETURN
1015 ON @SCREEN.OBJ.#,GOSUB 0,1025,0,0,1045,0,0,1060,0,0,1075,0
1020 RETURN
1025 IF @SCREEN.OBJ.VALUE[0,1] <> EnterKey THEN RETURN
1030 @SCREEN.OBJ.TEXT$[0,2] = @SCREEN.OBJ.TEXT$[0,1] : @SCREEN.OBJ.TEXT$[0,1] = ""
1035 IF @SCREEN.OBJ.TEXT$[0,2] = "exit" THEN 9999
1040 RETURN
1045 IF @SCREEN.OBJ.VALUE[0,4] <> EnterKey THEN RETURN
1050 @SCREEN.OBJ.TEXT$[0,5] = @SCREEN.OBJ.TEXT$[0,4] : @SCREEN.OBJ.TEXT$[0,4] = ""
1055 RETURN
1060 IF @SCREEN.OBJ.VALUE[0,7] <> EnterKey THEN RETURN
1065 @SCREEN.OBJ.TEXT$[0,8] = @SCREEN.OBJ.TEXT$[0,7] : @SCREEN.OBJ.TEXT$[0,7] = ""
1070 RETURN
1075 IF @SCREEN.OBJ.VALUE[0,10] <> EnterKey THEN RETURN
1080 @SCREEN.OBJ.TEXT$[0,11] = @SCREEN.OBJ.TEXT$[0,10] : @SCREEN.OBJ.TEXT$[0,10] = ""
1085 RETURN
9999 END
```

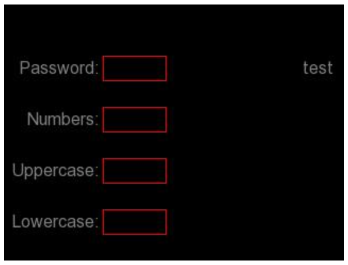
When the program is started this is the screen displayed:



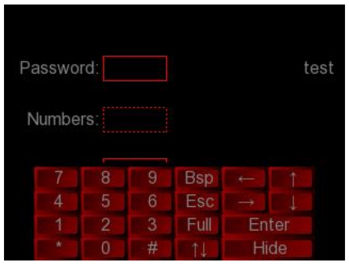
Touching the Password textbox pops up the keypad, and animates the textbox rectangle indicating that the control has focus. Pressed keys are entered in password style until the Enter key is pressed then the control’s .TEXT$ content is transferred to the label control on the same line and the textbox is cleared:

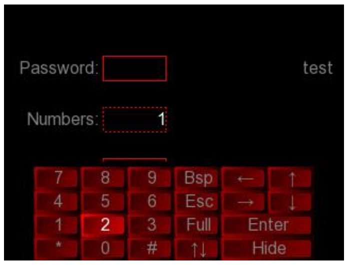


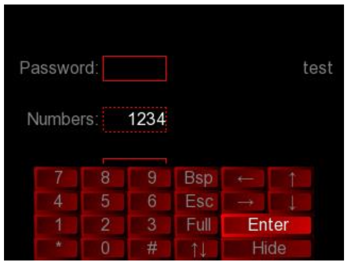


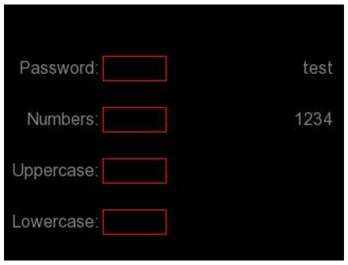


Touching the Number textbox pops up the numeric keypad and animates the textbox rectangle indicating that the control has focus. Pressed keys are entered until the Enter key is pressed then the control’s .TEXT$ content is transferred to the label control on the same line and the textbox is cleared:









@SCREEN.OBJ.SCHEME[screen, object]

This system variable sets the Scheme table entry of the scheme or lower entry of the Scheme pair that is used to render this screen object table entry.

@SCREEN.OBJ.X[screen, object]

This system variable sets the X coordinate of the position of the screen object relative to the screen for this screen object table entry.

@SCREEN.OBJ.Y[screen, object]

This system variable sets the X coordinate of the position of the screen object relative to the screen for this screen object table entry.

@SCREEN.OBJ.VALUE[screen, object]

This system variable sets or retrieves the value associated with the screen object for this screen object table entry The values are specific to each @SCREEN.OBJ.TYPE.

TBD – need descriptions of all of the Object Types and how their values interact here.

@SCREEN.OBJ.TOUCHED[screen, object]

This system variable sets or retrieves the touch state associated with the screen object for this screen object table entry.

@SCREEN.OBJ.IMAGE$[screen, object]

This system variable sets or retrieves the image resource used to render the screen object for this screen object table entry.

@SCREEN.OBJ.OVERLAY$[screen, object]

This system variable sets or retrieves the overlay image resource used to render the screen object for this screen object table entry.

@SCREEN.OBJ.TEXT$[screen, object]

This system variable sets or retrieves the optional text resource used to render the screen object for this screen object table entry.

@SCREEN.OBJ.OPTION[screen, object, option]

This system variable sets or retrieves the option values used to render the screen object for this screen object table entry. Here are the currently defined option values for the defined object types:

Icon Screen Object	
option	***function***
0	Number of Images override
1	Attributes: 1 Disable Touch Advance
2	
3	Horizontal Alignment: 0 Left 1 Center 2 Right
4	Vertical Alignment: 0 Bottom 1 Middle 2 Top
5	
6	
7	
8	
9	

Button Screen Object	
option	***function***
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Toggle Button Screen Object	
option	***function***
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Back Button Screen Object	
option	***function***
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Slider Screen Object	
option	***function***
0	
1	
2	
3	
4	
5	Minimum Value
6	Minimum Offset
7	Maximum Value
8	Maximum Offset
9	

Label Screen Object	
option	***function***
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Radial Gauge Screen Object	
option	*function*
0	Needle Length
1	Needle Zero
2	Needle Width
3	Needle Horizontal Alignment: 0 Left 1 Center 2 Right
4	Needle Vertical Alignment: 0 Bottom 1 Middle 2 Top
5	Minimum Angle
6	Minimum Value
7	Maximum Angle
8	Maximum Value
9	

Touch Keypad Screen Object	
option	*function*
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Linear Gauge Screen Object	
option	*function*
0	Needle Length
1	Needle Zero
2	Needle Width
3	Needle Horizontal Alignment: 0 Left 1 Center 2 Right
4	Needle Vertical Alignment: 0 Bottom 1 Middle 2 Top
5	Minimum Angle
6	Minimum Value
7	Maximum Angle
8	Maximum Value
9	

Listbox Screen Object	
option	*function*
0	Number of Data Items
1	Top Item
2	Item Height Override
3	
4	
5	
6	
7	
8	
9	

Spinner Knob Screen Object	
option	***function***
0	Indicator Length
1	Indicator Zero
2	Indicator Width
3	
4	
5	Minimum Angle
6	Minimum Value
7	Maximum Angle
8	Maximum Value
9	

Textbox Screen Object	
option	***function***
0	Styles: 1 Numbers Only 2 Password (only '*' displayed) 4 Lowercase 8 Uppercase 16 Read Only
1	Maximum Number of Characters
2	
3	
4	
5	
6	
7	
8	
9	

@SCREEN.OBJ.#

This read-only system variable reflects the screen object number of the last screen object event.

@SCREEN.OBJ.EVENT

This system variable reflects the last screen object event. There are currently three screen object events:

@SCREEN.OBJ.EVENT	Event
1	Screen Object[@SCREEN.#, @SCREEN.OBJ.#] was touched
2	Screen Object[@SCREEN.#, @SCREEN.OBJ.#] value has changed
3	Screen Object[@SCREEN.#, @SCREEN.OBJ.#] was un-touched (released)

@SCREEN.OBJ.MASK

This system variable gets or sets the screen object event mask. There is one bit to enable or disable each of the screen object events – a set bit enables the event. The default @SCREEN.OBJ.MASK is all events enabled:

Where the bits are numbered:

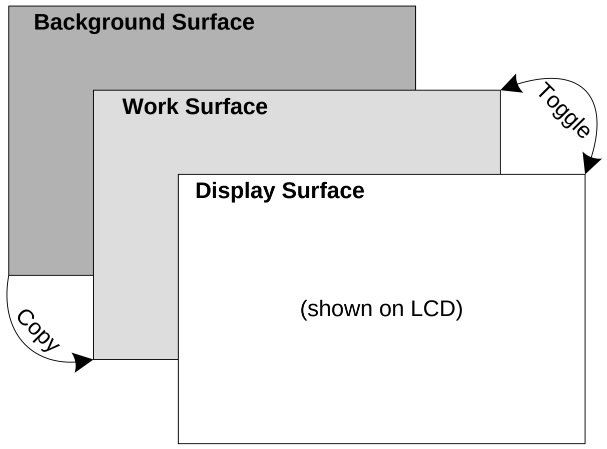
15 MSB	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0 LSB**

Bit #	# Value	@SCREEN.OBJ.EVENT
0	1	Screen object touched
1	2	Screen object value changed
2	4	Screen object un-touched (released)

@SURFACE

This system variable sets the drawing surface for subsequent DRAW. commands. There are three drawing surfaces:

@SURFACE	Drawing Surface
0	Display
1	Work
2	Background



@TOUCH.

These system variables allow program interaction with the display touchscreen and pop-up keypad:

@TOUCH.KEYPAD

This system variable is event capable, is fired whenever a touch keypad event occurs and gets or sets the pop-up touch keypad status:

@TOUCH.KEYPAD	Keypad
0	no keypad
1	QWERTY keypad
2	Numeric keypad
3	QWERTY keypad top
4	Numeric keypad top
5	Configuration keypad

@TOUCH.EVENT

This system variable is event capable and is fired whenever a touchscreen event occurs. The event values are:

@TOUCH.EVENT	Event
0	no event
1	touchscreen press event
2	touchscreen move event
3	touchscreen release event

@TOUCH.X

This system variable holds the X coordinate of the @TOUCH.EVENT

@TOUCH.Y

This system variable holds the Y coordinate of the @TOUCH.EVENT

Operators

ACS Basic supports the following operators listed in priority from highest to lowest. Operators encountered during statement execution are evaluated in order of priority with higher priority operators executed before lower priority operators.

Operators work between a left and right operand – unary operators only work on a right, following operand.

Operator	Description	Priority
`NOT`	Logical NOT	7
`-`	Unary minus (negate, 2's complement)	7
`~`	Unary Bitwise NOT (1's complement)	7
`*` `/` `%`	Multiplication, division, modulus	6
`+`	Addition, string concatenation	5
`-`	Subtraction	5
`<<` `>>`	Left Shift, Right Shift	4
`=` `<>`	Assign / test equal, test NOT equal (numeric or string)	3
`<` `<=` `>` `>=`	Test Less Than, Less than or Equal, Greater Than, Greater than or Equal (numeric or string)	3
`&` `|` `^`	Bitwise AND, OR, Exclusive OR	2
`AND` `OR`	Logical AND, OR	1

Parenthesis may be used to change or enforce expression execution priority with the innermost grouped parenthesis expression evaluated first.

The six 'test' relational operators **(=, <>, <, <=, >, >=)** can be used in any expression, and evaluate to 1 if the tested condition is TRUE, and 0 if it is FALSE. The IF and LIF commands accept any non-zero value to indicate a TRUE condition.

Multiple 'test' operators can be combined with the logical NOT, AND, OR operators and suitable parenthesis.

There are six operators for bit manipulation (**~, &,** |, ^, <<, >>); these may only be applied to integer operands. The 16 'bit' positions in the integer are numbered from right to left starting with 0 (the **L**east **S**ignificant **B**it) up to 15 (the **M**ost **S**ignificant **B**it) or sign bit:

MSB															**LSB**
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
16-bit Integer value															

Thus the value 1234 in binary bit form is:

Decimal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1234	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0

And the value -1234 in binary bit form is:

Decimal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-1234	1	1	1	1	1	0	1	1	0	0	1	0	1	1	1	0

The bitwise ~ unary operator yields the one's complement of its following integer operand; that is, it converts each 1-bit into a 0-bit and vice versa. Thus the value ~1234 in binary bit form is:

Decimal	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
~1234	1	1	1	1	1	0	1	1	0	0	1	0	1	1	0	1

Note that each bit position in the ~1234 is inverted from their 1234 values.

The bitwise **&** operator is often used to mask off or clear some set of bits. This can be used to determine which bits are set by &'ing a value with the mask of the bit to examine. So the value 1234 bitwise **&** with 255 is 210:

Decimal	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1234	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0
& 255	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
= 210	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1	0

The bitwise **|** operator is used to turn on or set some set of bits. So the value 1234 bitwise **|** with 255 is 1279:

Decimal	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1234	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0
| 255	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
= 1279	0	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1

The bitwise exclusive or operator ^ sets a one in each bit position where its operands have different bits, and zero where they are the same. This can be used to toggle specific bits by ^'ing a value with the bits to toggle. So the value 1234 ^ with 255 is 1069:

Decimal	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1234	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0
^ 255	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
= 1069	0	0	0	0	0	1	0	0	0	0	1	0	1	1	0	1

The bitwise << and >> perform left and right shifts of their left operand by the number of bit positions given by their right operand, which must be positive. Vacated bits on the right are filled by zeroes, vacated bits on the left are filled with the value of the sign bit.

The bitwise << shifts the bits towards the left from LSB towards MSB, filling in the vacated LSB positions with zero bits. Thus 1234 << 2 = 4936:

Decimal	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	
1234	0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0	
<< 2	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	←	← 0, 0
= 4936	0	0	0	1	0	0	1	1	0	1	0	0	1	0	0	0	

The bitwise >> shifts the bits towards the right from MSB towards LSB, filling in the vacated MSB positions with copies of the sign bit 15. Thus1234 >> 2 = 308:

Decimal		**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1234		0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0
>> 2	0, 0 →	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→
= 308		0	0	0	0	0	0	0	1	0	0	1	1	0	1	0	0

Since the bits filling into the vacated MSB positions are copies of the sign bit, bit 15 then -1234 >> 2 = -308:

Decimal		**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-1234		1	1	1	1	1	0	1	1	0	0	1	0	1	1	1	0
>> 2	1, 1 →	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→
= -308		1	1	1	1	1	1	1	0	1	1	0	0	1	0	1	1

Expressions

In ACS Basic expressions consist of one or more variables, constants, functions or system variables that may optionally be joined together by Operators. The evaluation order may be controlled by the judicious use of parenthesis. Expressions may be nested up to 10 levels. Some examples:

```
a=10
Ready
print a*30
300
Ready
print fmt$("%02X", a)
0A
Ready
print a<<2
40
Ready
print (a<<2)=0
0
Ready
print (a<<2)<>0
1
Ready
print a^4
14
Ready
```

Functions

ACS Basic provides several functions that may be used in expressions. There must not be a space between the function name and the opening parenthesis. Functions must be used in a statement such as a LET or PRINT – they cannot be executed standalone in immediate mode.

ASC(char)

Returns the numeric ASCII value of the character argument.

```
PRINT ASC("A")
65
```

ABS(expr)

Returns the absolute value of the numeric argument.

```
PRINT ABS(10), ABS(-10)
10 10
```

CHR$(expr)

Returns an ASCII string containing the character equivalent of the expression argument.

```
PRINT CHR$(65)
A
```

COS(degrees)

Returns a scaled sine value of the degree argument where $-1024 \leq \text{COS( )} \leq 1024$. The degree argument ranges from $0 \rightarrow 360$ and arguments larger than 360 degrees are converted modulo 360.

COS(0) = 1024, COS(90) = 0, COS(180) = -1024, COS(270) = 0, etc..

ERR( )

Returns the last error number.

ERR$( )

Returns the string representation of the last error number.

FILE.EXISTS("path")

Returns one if the file specified by the "path" argument exists, otherwise zero.

FIND(var$, searchvar$ {, startpos})

Returns the zero based position of string **search var**iable in string **var**iable starting at zero (or optional **startpos)** or -1 if the search variable was not found.

```
TEST$="012345" : PRINT FIND(TEST$, "3")
3
TEST$="012345" : PRINT FIND(TEST$, "3", 4)
-1
```

FMT$(fmt$ {, expr{$}, expr{$} … , expr{$}}})

Returns a formatted ASCII string of zero or more numeric or string **expr**essions using the string format specification **fmt$**.

A format specification string consists of zero or more **{**optional**}** and required fields and has the following form:

%{Flags}{Width}{.Precision}Type

Each field of a format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a **type** character (for example, %d). If a percent sign is followed by a character that has no meaning as a format field, the character is copied to the return value. For example, to produce a percent sign in the return value, use %%.

The optional fields, which appear before the **type** character, control other aspects of the formatting, as follows:

Type	Required character that determines whether the associated *argument* is interpreted as a character, a string, or a number:	
	c	character
	d	signed decimal integer
	i	signed decimal integer
	u	unsigned decimal integer
	s	string
	o	unsigned octal integer
	x	unsigned hexadecimal integer
	X	unsigned HEXADECIMAL integer

Flags	Optional character or characters that control justification of output and printing of signs, blanks, and octal and hexadecimal prefixes. More than one flag can appear in a format specification. - : left align the result in the given field width + : prefix the output with a sign (+/-) if the type is signed 0 : if **Width** is prefixed with 0, zeroes are added until the minimum width is reached. If 0 and – appear, the 0 is ignored. If 0 is specified with an integer format, the 0 is ignored. *blank(' ')* : prefix the output with a blank if the result is signed and positive; the blank is ignored if both the blank and + flags appear # : when used with o, x or X format, prefix any nonzero output value with 0, 0x or 0X respectively, otherwise ignored
Width	Nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified width, blanks are added to the left or the right of the values — depending on whether the – flag (for left alignment) is specified — until the minimum width is reached. If **Width** is prefixed with 0, zeroes are added until the minimum width is reached (not useful for left-aligned numbers). The **Width** specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if **Width** is not given, all characters of the value are printed (subject to the **Precision** specification).
Precision	Specifies a nonnegative decimal integer, preceded by a period (**.**), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits. Unlike the **Width** specification, the precision specification can cause truncation of the output value. If **Precision** is specified as 0 and the value to be converted is 0, the result is no characters output. c : **Precision** has no effect d,i,u,o, x,X : **Precision** specifies the minimum number of digits to be output. If the number of digits is less than **Precision**, the output is padded on the left with zeroes. The value is not truncated when the number of digits exceeds **Precision** s : **Precision** specifies the maximum number of characters to be output. Characters in excess of **Precision** are not output

```
10 REM show time
15 ONEVENT @SECOND,GOSUB 100
20 GOTO 20
100 PRINT FMT$("%c%2d:%02d:%02d",13,@HOUR,@MINUTE,@SECOND);
105 RETURN
Ready
run
13:30:13 <<< ESC at line 20 >>>
Ready
```

GETCH(expr)

If ***expr*** evaluates to zero, **GETCH(0)** returns the numeric value of the next available serial character (if **@MSGENABLE**=0) or it returns a -1 if no character is currently available from either enabled source.

If ***expr*** evaluates to non-zero, **GETCH(1)** waits for the next available serial character (if **@MSGENABLE**=0) and then returns its numeric value.

HEX.STR$(expr{,digits})

Returns a string containing the hexadecimal representation of the ***expr***ession. The optional digits parameter specifies the number of digits.

```
PRINT HEX.STR$(43690)
AAAA
Ready
PRINT HEX.STR$(43690,6)
00AAAA
Ready
```

HEX.VAL(var$)

Returns the numeric value of the hexadecimal string ***var$*** variable.

```
PRINT HEX.VAL("AAAA")
43690
Ready
```

INSERT$(var$, start, var2$)

Returns a string ***var***iable with the contents of ***var***iable***2*** inserted at zero based position ***start***.

```
10 REM test insert$
20 s$ ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 i$ ="insert"
35 REM insert at beginning
40 PRINT INSERT$(s$,0,i$)
45 REM insert in middle
50 PRINT INSERT$(s$,13,i$)
55 REM insert past end
60 PRINT INSERT$(s$,30,i$)
Ready
run
insertABCDEFGHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMinsertNOPQRSTUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZinsert
Ready
```

LEFT$(var$, len)

Returns a string containing the leftmost **len**gth characters of string **var**iable.

```
TEST$="This is a string" : PRINT LEFT$(TEST$,5)
This
```

LEN(var$)

Returns the length (number of characters) of string **var**iable.

```
TEST$="This is a string" : PRINT LEN(TEST$)
16
```

MID$(var$, start, len)

Returns a string consisting of **len**gth number of characters of string **var**iable from zero based ***start*** character position.

```
TEST$="This is a string" : PRINT MID$(TEST$,5,4)
is a
```

MULDIV(number, multiplier, divisor)

Returns a 32 bit result of ((number * multiplier) / divisor) where number, multiplier and divisor are 64-bit internally. Useful for calculating percentages, etc., where the normal multiply would overflow a signed 32-bit number.

```
10 REM calculate 55 percent of 999
20 PRINT MULDIV(999,55,100);".";MULMOD(999,55,100)
Ready
run
549.45
```

MULMOD(number, multiplier, divisor)

Returns a 32 bit result of ((number * multiplier) % divisor) where number, multiplier and divisor are 64-bit internally. Useful for calculating remainders of percentages, etc., where the normal multiply would overflow a signed 32-bit number.

RIGHT$(var$, len)

Returns a string containing the rightmost **len**gth characters of string **var**iable.

```
TEST$="This is a string" : PRINT RIGHT$(TEST$,6)
string
```

REPLACE$(var$, start, var2$)

Returns a string ***var***iable with the contents of ***var***iable***2*** overwritten at zero based position ***start***.

```
10 REM test replace$
20 s$ ="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
30 r$ ="replace"
35 REM replace at beginning
40 PRINT REPLACE$(s$,0,r$)
45 REM replace in middle
50 PRINT REPLACE$(s$,13,r$)
55 REM replace past end
60 PRINT REPLACE$(s$,30,r$)Ready
run
replaceHIJKLMNOPQRSTUVWXYZ
ABCDEFGHIJKLMreplaceUVWXYZ
ABCDEFGHIJKLMNOPQRSTUVWXYZreplace
Ready
```

RND(expr)

Returns a psuedo random number that ranges from 0 to (**expr**ession - 1).

```
10 FOR i= 0 TO 10 : PRINT RND(10);" "; : NEXT i
Ready
run
3 0 4 3 8 3 1 1 3 4 3 Ready
```

SIN(degrees)

Returns a scaled sine value of the degree argument where $-1024 \leq \text{SIN}(\ ) \leq 1024$. The degree argument ranges from $0 \rightarrow 360$ and arguments larger than 360 degrees are converted modulo 360.

SIN(0) = 0, SIN(90) = 1024, SIN(180) = 0, SIN(270) = -1024, etc..

STR$(expr)

Returns a string representation of the numeric argument.

```
TEST$ = STR$(1234) : PRINT TEST$
1234
```

SOCKET.SYNC.CONNECT(#N, "ip:port", connect( ), send( ), recv( ))

Initiates an outgoing synchronous network socket connection as file #N using the string representation of the IPv4 IP address and port number. The status of the connection, send, receive and disconnect process is returned as the value of this numeric function which can return the following values:

SOCKET.SYNC.CONNECT( ) returns	**Description**
0	Unknown / No Status
1	Disconnect / Done / No Error
2	Open Error – requested socket failed to open
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT[#N]** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT[#N]** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT[#N] interval**

- The **connect( )** user function is called when a connection is established.
- The **send( )** user function is called to send data to the connected device using PRINT #N or FPRINT #N statement(s). Return zero to terminate the connection, one to proceed to the **recv( )** function or two to be called again to send more data.
- The **recv( )** user function is then called to receive data from the connected device using INPUT #N or FINPUT #N statements(s). Return zero to terminate the connection, one to return to the **send( )** function and two to be called again to receive more data.

See the Socket Programming section below for more information and sample programs.

SOCKET.SYNC.LISTEN(#N, ":port", connect( ), recv( ), send( ))

Initiates an incoming synchronous network socket reception as file #N using the string representation of the IPv4 IP port number. The status of the connection, send, receive and disconnect process is returned as the value of this numeric function which can return the following values:

SOCKET.SYNC.CONNECT( ) returns	**Description**
0	Unknown / No Status
1	Disconnect / Done / No Error
2	Open Error – requested socket failed to open
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT[#N]** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT[#N]** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT[#N]** interval

- The **connect( )** user function is called when a connection is established.
- The **recv( )** user function is then called to receive data from the connected device using INPUT #N or FINPUT #N statements(s). Return zero to terminate the connection, one to return to the **send( )** function and two to be called again to receive more data.
- The **send( )** user function is called to send data to the connected device using PRINT #N or FPRINT #N statement(s). Return zero to terminate the connection, one to proceed to the **recv( )** function and two to be called again to send more data.

See the Socket Programming section below for more information and sample programs.

UBOUND(dimVariable{[dimNumber]})

Returns the size of the **dimVariable** as it was declared in the **DIM** statement. The optional **dimNumber** in square brackets defaults to 0 and can be 0, 1 or 2 to obtain the size of the corresponding dimension.

```
10 REM test multidimensional arrays
15 DIM test[3,4,5]
20 FOR x = 0 TO UBOUND(test[0])-1
25  FOR y = 0 TO UBOUND(test[1])-1
27   FOR z = 0 TO UBOUND(test[2])-1
30    test[x,y,z] = x * y + z
32   NEXT z
35  NEXT y
40 NEXT x
45 PRINT "test[";UBOUND(test[0]);",";UBOUND(test[1]);",";UBOUND(test[2]);"] ="
50 FOR x = 0 TO UBOUND(test[0])-1
55  FOR y = 0 TO UBOUND(test[1])-1
57   FOR z = 0 TO UBOUND(test[2])-1
60    PRINT test[x,y,z];",";
62   NEXT z
63   PRINT ""
65  NEXT y
67  PRINT ""
70 NEXT x
Ready
run
test[3,4,5] =
0,1,2,3,4,
0,1,2,3,4,
0,1,2,3,4,
0,1,2,3,4,

0,1,2,3,4,
1,2,3,4,5,
2,3,4,5,6,
3,4,5,6,7,

0,1,2,3,4,
2,3,4,5,6,
4,5,6,7,8,
6,7,8,9,10,

Ready
```

VAL(expr$)

Returns the numeric value of the string argument representation of a number.

```
TEST = VAL("1234") : PRINT TEST
1234
```

Events

ACS Basic provides the concept of an ***Event***. Events occur outside of the normal program execution flow and are processed in between the execution of individual program statements. Some system variables have ***Events*** associated with them and may be referenced in **ONEVENT, SIGNAL** and **WAIT** statements. There are three ways to process an event: asynchronously with an **ONEVENT** handler, synchronously with a **WAIT** statement or by polling the system variable's value in the program to see when the event occurs.

In order to process an event asynchronously, Basic has to be informed of what code to execute when a certain event occurs. This is done using the **ONEVENT** statement. After Basic executes each program statement, it scans the table of events looking to see if any have been signaled. If an **ONEVENT** handler for a signaled event has been specified by the program, then Basic will force a subroutine call to the event handler before the next program statement is executed.

Events have an implicit priority with higher priority events being able to interrupt execution of lower priority event handlers. Here's an example of touch event handling:

```
100 REM Test Touch Events
110 ONEVENT @TOUCH.EVENT, GOSUB 1000
130 GOTO 130
1000 T=@TOUCH.EVENT: X=@TOUCH.X: Y=@TOUCH.Y
1010 ON T, GOTO 1015,1020,1025,1030
1015 RETURN
1020 PRINT "Touch   @ ";X;", ";Y : RETURN
1025 PRINT "Move    @ ";X;", ";Y : RETURN
1030 PRINT "Release @ ";X;", ";Y : RETURN
1035 RETURN
Ready
```

This would print **"Touch @ x, y", "Move @ x, y", or "Release @ x, y"** on the screen as the user interacts with the touchscreen.

In order to handle an event synchronously a program may wait for an event to occur by using the **WAIT** statement. Program execution stalls at that statement until the specified event happens. Alternatively, the program may poll the associated system variable's value in a loop looking for the event to have been signaled. Here's an example of polling for the @SECOND system variable to change:

```
10 REM poll @SECOND
15 Seconds = @SECOND
20 LIF Seconds <> LastSeconds THEN PRINT Seconds : LastSeconds = Seconds
25 GOTO 15
Ready
run
54
55
56
57
 ESC at line 15
Ready
```

This would print the value of @SECOND every time it changes.

The **SIGNAL** statement may be used in a program to force an event to happen.

<u>It is very important to note that the **ONEVENT** handler subroutine executes in the context of the running program: it has access to all program variables. Since the event handler may be executed at any time in between any program statements care should be used when changing program variables from within an event handler as it may cause unexpected results in the execution of other program statements that may be using and depending upon the values of those same variables. Incorrect or unexpected program execution may result – code event handlers carefully.</u>

See the **ONEVENT** statement definition below for a table showing what events may be processed and listing their relative priority.

Statements

ACS Basic program lines consist of an optional integer line number followed by one or more statements. Multiple statements on a line are allowed, separated by a colon (‘**:**’). Only the first statement on a line may have a line number. A Direct mode of operation is available for some statements when they are entered without a line number and are executed immediately. Here are some sample program statements:

```
10 REM This is a comment
20 FOR I=0 TO 10:PRINT I:NEXT I
```

The statement keywords are ‘tokenized’ when entered to save program memory and speed program execution: *ie:* the keyword **GOSUB** would be tokenized to a single byte instead of five bytes. In addition, the statement line numbers are converted to a four-byte unsigned integer form to save space and facilitate program execution. Saved programs are expanded (un-tokenized) on the SD card to allow program storage, viewing and editing with an external text editor if required.

The following statement keywords are supported:

BREAK {line} / BREAK {`label}

Program mode only. Exits from within **FOR / NEXT** or **WHILE / WEND** loops – execution continues after the closest **NEXT** or **WEND** statement. If the optional **line** or **`label** is present execution continues there.

```
10 REM for/break/next test
15 FOR i = 0 TO 10
20  IF i > 5 THEN BREAK
25  PRINT i
30 NEXT i
Ready
run
0
1
2
3
4
5
Ready
```

```
10 REM for/break/next test
15 FOR i = 0 TO 10
20  IF i > 5 THEN BREAK `out
25  PRINT i
30 NEXT i
35 END
40 `out : PRINT "Done"
Ready
run
0
1
2
3
4
5
Done
Ready
```

CHANGE string, replacement

Searches the entire program for <u>case sensitive</u> match of **string** and then prompts Yes/No/All/eXit for **replacement**. If either string or replacement contains a space enclose both in double-quotes and separate them with a comma:

```
change send_200 send_200_header
15 CONST send_200=0, send_content=1, send_file=2, send_404=3, send_disconnect=4
         ^ = send_200_header (Yes/No/All/eXit) ?y
15 CONST send_200_header=0, send_content=1, send_file=2, send_404=3, send_disconnect=4
40 recvdata$="" : senddata$="" : sendstate=send_200 : line=0
                                              ^ = send_200_header (Yes/No/All/eXit) ?a
40 recvdata$="" : senddata$="" : sendstate=send_200_header : line=0
1150 line=0:ONERROR GOTO 1152 : sendstate=send_404 : OPEN #1, filename$,"r" : sendstate=send_200_header : ONERROR GOTO 0
Ready
```

CLEAR

Erases all variables and closes all open files.

CLOSE #N

Close file or internet streaming handle #*N* (0 → 9) opened with **OPEN #N** statement.

CONST var{$}=value {, var{$}=value … }

Creates one or more constant variables that can't be assigned to after they are created.

```
10 REM define CONSTants
20 CONST Limit=10, Abort$="ABORT!"
Ready
run
Ready
vars
Limit  -> r/o Int          = 10
Abort$ -> r/o Str$         = "ABORT!"
Ready
```

CONTINUE

Program mode only. Continues next iteration of **FOR / NEXT** or **WHILE / WEND** loops - execution continues at the closest **NEXT** or **WHILE** statement.

```
10 REM while/continue/wend test
15 WHILE a < 10
20  a = a + 1
25  IF (a & 1) THEN CONTINUE
40  PRINT a
45 WEND
Ready
run
2
4
6
8
10
Ready
```

DATA

Program mode only. Enter "inline" **DATA** statements holding values that can be accessed by **READ** and **ORDER** statements. All related **DATA** statements should be in a group of sequential lines.

DEL path

Delete files and directories on the SD card. The full ***path*** must be specified. Directories must be empty to be deleted. In program mode ***Path*** may be a constant string or you can use a string variable as the ***path*** by concatenating it to such a string: **DEL ""+PATH$**. In direct mode the quotation marks are not required.

DELAY value

Pause program execution for value * 20mSEC. While the delay is in process, Events can occur but any defined **ONEVENT** handlers will not be executed until the delay has expired.

```
10 REM delay for one second
20 DELAY 50
```

DIM var{$} [size{, size1{, size2}}]

Dimension a numeric or character array **var**iable to hold **size** integers or character strings. Arrays may be defined with up to three dimensions shown as **size**, **size1** and **size2**. Array variable elements may then be accessed using numeric indices separated by commas and enclosed in square brackets that range from the first element of zero to the last element of each dimension: A[0], A[1], … , A[size - 1]. If an attempt is made to access a variable as an array before it has been dimensioned a "Dimension Error" will result. If an attempt is made to access an array element with a negative index or an index beyond the currently defined array size an "Index Out of Range Error" will result. A variable may be re-dimensioned, however the current contents of the variable will be lost.

```
10 REM multidimensional arrays
15 DIM test[3,4,5]
20 FOR x = 0 TO 2
25  FOR y = 0 TO 3
27   FOR z = 0 TO 4
30    test[x,y,z] = x * y + z
32   NEXT z
35  NEXT y
40 NEXT x
Ready
```

DIR {path}

Show files on the SD card. An optional ***path*** may be specified. Wildcard characters '?' and '*' may be used in the ***path*** with '?' matching any single character and/or '*' matching multiple characters to show multiple files matching the pattern.

```
dir s*.wav
S7DB.WAV   323,028 A      08-08-2013 12:36:48 PM
S7EB.WAV   306,060 A      05-04-2006 12:39:00 PM
S7FB.WAV 1,164,232 A      05-04-2006 12:42:00 PM
-----------------
                 3 file(s)      1,793,320 bytes
                 0 dir(s)   7,929,528,320 bytes free
Ready
```

DIR #N, {path}

Write a list of files on the SD card to an open file #***N*** (0 → 9). An optional ***path*** may be specified. Wildcard characters '?' and '*' may be used in the ***path*** with '?' matching any single character and/or '*' matching multiple characters to show multiple files matching the pattern.

EDIT line

Direct mode only. Using an ANSI terminal allows editing a line by displaying the statement, moving the cursor with the Home, Left arrow, Right arrow and End keys. The Backspace key is used to delete characters to the left of the cursor. Typed characters are entered at the cursor. The Enter key accepts the changes, a double ESC key aborts the edit.

END

Program mode only. Terminate program with no message. Closes all open files.

ERROR value

Force an error. Program execution stops and an error message is displayed.

```
10 ERROR 250
```

```
Ready
run
250 error in line 10
Ready
```

FOR var=init TO limit {STEP increment} : statements : NEXT var

Program mode only. Performs a counted loop; incrementing ***var*** from the ***init*** expression value to the ***limit*** expression value by the optional ***increment*** expression value, executing statements up until the matching **NEXT** statement. The default value for the optional **STEP increment** is one. When the **NEXT** statement is reached execution resumes with the matching **FOR** statement if the **STEP increment** of the control **var**iable has not reached the **limit**.

- **FOR / NEXT** loops can be nested and don't have to be the only statements on the line.
- **FOR / NEXT** loops can be exited from within without the ***var***iable reaching the ***limit*** using the **BREAK** statement.
- **FOR / NEXT** loops can be continued from within without executing all of the loop statements using the **CONTINUE** statement.
- As the **FOR / NEXT** counted loop uses the control stack to execute you should not jump out of or into the encompassing code block of statements. To leave the counted loop force the ***var***iable to the ***limit*** value or use the **BREAK** statement.
- The number of nested **FOR / NEXT, WHILE / WEND** loops, block **IF / THEN / ELSE / ENDIF, FUNCTION / ENDFUNCTION** and **GOSUB** subroutines is limited.
- Execution of a **FOR** statement without a subsequent **NEXT** causes a "Nesting Error".
- Execution of a **NEXT** statement without a preceding **FOR** causes a "Nesting Error".

FINPUT #N, var{$}, … , var{$}

Gets the value(s) for one or more **var**iables from a single line from open file #*N* (0 → 9). Note that when an end of file occurs, the **var**iables will have their last value. Test the **@FEOF[#N]** system variable to detect this condition. The data items in the file are separated by commas, and string values must be surrounded by double quotes. See the **FPRINT #N** statement below that can be used to produce a file in the correct format. If the data in the file ends before all of the variables have been assigned values an "Out of Data Error" occurs. Incorrect data formatting in the file can cause a "Syntax Error" to occur.

```
10 OPEN #0, "test.csv", "r"
20 LIF @FILE.SIZE[#0] = 0 THEN PRINT "empty file" : END
30 FINPUT #0, Number, Number$
40 LIF @FEOF[#0] = 0 THEN PRINT Number, Number$ : GOTO 30
Ready
type test.csv
0,"zero"
1,"one"
2,"two"
Ready
run
0 zero
1 one
2 two
Ready
```

FPRINT #N, expr{, expr…}

Prints one or more expression(s) to the file #N (0 → 9) that is **OPEN**ed for writing as a single line. The data items on the line in the file are separated by commas, with string values surrounded by double quotes. The produced file is compatible with the **FINPUT #N** statement.

```
10 OPEN #0, "test.csv", "w+"
20 FPRINT #0, 0, "zero" : FPRINT #0, 1, "one" : FPRINT #0, 2, "two"
30 CLOSE #0
run
Ready
Type test.csv
0,"zero"
1,"one"
2,"two"
Ready
```

FOPEN #N, recordlength, ”path”

Opens filename ***path*** as a fixed record length file #***N*** (0 → 9) for subsequent sequential / random access via **FREAD# / FWRITE#** statements.

- If ***recordlength*** is negative or greater than 255 it is forced to 255.
- The ***recordlength*** includes the trailing CR/LF character pair that terminates each record.
- If the file is empty, **@FEOF[#N]** will be set.

FREAD #N, recordnumber, var{$}, var{$}, … var{$}

Reads ASCII data from fixed length records on file ***#N*** (0 → 9) opened by **FOPEN #N** into the list of variables. Before the data is read, the file is positioned to the desired ***recordnumber*** by positioning the file to (***recordnumber*** x ***recordlength***). ***(***0 ≤ ***recordnumber*** x ***recordlength*** ≤ 2,147,483,648***)***. A negative ***recordnumber*** seeks to the end of the file.

- Reading at the current end of the file sets the **@FEOF[#N]** system variable and signals the associated event. Note that when an end of file occurs, the **var**iables will have their last value from a prior successful **FREAD**.
- Reading past the current end of the file generates a “FREAD record # Out of Range error”.
- The data items in the file are separated by commas, with string values surrounded by double quotes. If the data in the file ends before all of the variables have been assigned values an “Out of Data Error” occurs. Incorrect data formatting in the file can cause a “Syntax Error” to occur.

```
10 LIF FILE.EXISTS("test.dat") = 0 THEN PRINT "no test.dat" : END
20 FOPEN #1,20,"test.dat"
30 r = 0 : WHILE @FEOF[#1] = 0
40  FREAD #1,r,Number,String$
50  LIF @FEOF[#1] = 0 THEN PRINT Number, String$ : r = r + 1
60 WEND
70 CLOSE #1
Ready
type test.dat
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
run
0 str0
1 str1
2 str2
3 str3
4 str4
```

```
5 str5
6 str6
7 str7
8 str8
9 str9
Ready
```

FWRITE #N, recordnumber, var{$}, var{$}, ... var{$}

Writes ASCII data into fixed length records on file ***#N*** (0 → 9) opened by ***FOPEN #N*** from the list of variables. Before the data is written, the file is positioned to the desired ***recordnumber*** by positioning the file to (***recordnumber*** x ***recordlength***). ***(***0 ≤ ***recordnumber*** x ***recordlength*** ≤ 2,147,483,648***)***.

- A negative ***recordnumber*** seeks to the current end of the file.
- Writing at the current end of file extends the file by the record size. Writing past the current of file generates a "FWRITE record # Out of Range error".
- The data items written to the file are separated by commas, with string values surrounded by double quotes.
- The record is padded with spaces to ***recordlength*** including the trailing CR/LF character pair which terminates each record.
- The file may be viewed using the **TYPE** command.

```
10 IF FILE.EXISTS("test.dat") THEN DEL "test.dat"
15 FOPEN #1,20,"test.dat"
20 FOR r = 0 TO 9 : FWRITE #1, r, r, "str"+STR$(r) : NEXT r
25 CLOSE #1
Ready
run
Ready
type test.dat
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
```

FINSERT #N, recordnumber, var{$}, var{$}, ... var{$}

Inserts ASCII data into fixed length records on file ***#N*** (0 → 9) opened by ***FOPEN #N*** from the list of variables using a temporary file FINSERT.TMP. Before the data is inserted, the file is positioned to the desired ***recordnumber*** by positioning the file to (***recordnumber*** x ***recordlength***). ***(***0 ≤ ***recordnumber*** x ***recordlength*** ≤ 2,147,483,648***)***, and records in the file after ***recordnumber*** are shifted down.

- A negative ***recordnumber*** seeks to the end of the file before inserting.
- The data items inserted into the file are separated by commas, with string values surrounded by double quotes.
- The record is padded with spaces to ***recordlength*** including the trailing CR/LF character pair which terminates each record.
- The file may be viewed using the **TYPE** command.

```
type test.dat
```

```
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
list
10 LIF FILE.EXISTS("test.dat") = 0 THEN PRINT "no test.dat" : END
20 FOPEN #1,20,"test.dat"
25 FINSERT #1,0,-1,"str"+STR$(-1) : FINSERT #1,-1,10,"str"+STR$(10)
30 r = 0 : WHILE @FEOF[#1] = 0
40  FREAD #1,r,Number,String$
50  LIF @FEOF[#1] = 0 THEN PRINT Number, String$ : r = r + 1
60 WEND
70 CLOSE #1
Ready
run
-1 str-1
0 str0
1 str1
2 str2
3 str3
4 str4
5 str5
6 str6
7 str7
8 str8
9 str9
10 str10
Ready
```

FDELETE #N, recordnumber

Removes fixed length record ***recordnumber (***0 ≤ ***recordnumber*** x ***recordlength*** ≤ 2,147,483,648***)*** on file ***#N*** (0 → 9) opened by **FOPEN #N** using a temporary file FDELETE.TMP.

```
type test.dat
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
list
10 LIF FILE.EXISTS("test.dat") = 0 THEN PRINT "no test.dat" : END
20 FOPEN #1,20,"test.dat"
25 FDELETE #1,0
30 r = 0 : WHILE @FEOF[#1] = 0
40  FREAD #1,r,Number,String$
50  LIF @FEOF[#1] = 0 THEN PRINT Number, String$ : r = r + 1
60 WEND
70 CLOSE #1
Ready
run
1 str1
2 str2
3 str3
4 str4
5 str5
6 str6
7 str7
```

```
8 str8
9 str9
Ready
```

FUNCTION name{$}({parm1{$}{,parm2{$}, … parmN{$}}})

Program mode only. Defines a user function of **name{$}**. Functions may be either integer or string using the variable naming convention of a trailing dollar sign for strings. Functions require zero or more integer or string parameters enclosed in a parenthesized parameter list.

- Functions are defined by statements beginning with a **FUNCTION** command and ending with an **ENDFUNCTION** command.
- Statements following the **FUNCTION** command on the same line are not executed.
- Functions may be redefined using the same function **name{$}** as long as the integer or string function type is not changed.
- The function **name{$}** behaves like a global integer or string variable that has a zero or empty string value when the function is defined and can be used to provide a value to or return a value from the function.
- A defined function is executed using the function's name in a command or expression.

See the User Functions section below for more information.

ENDFUNCTION

Program mode only. Ends a user defined function. Statements following the **ENDFUNCTION** command on the same line are not executed. When initially defining a **FUNCTION** the **ENDFUNCTION** statement terminates the definition. When executing a previously defined **FUNCTION** the **ENDFUNCTION** causes program execution to return to the statements following the function call.

GOSUB line / GOSUB `label

Program mode only. Calls a subroutine that starts at ***line*** or ***`label*** and ends with a **RETURN** statement. A subroutine consists of a group of program statements that start at a certain ***line*** number or ***`label*** and end in a line with a **RETURN** statement.

- To call the subroutine from your program use the **GOSUB** statement which transfers program execution to the specified line number and executes those program statements until it executes a **RETURN** statement.
- Upon execution of the **RETURN** statement, program execution continues at the statement after the **GOSUB**.
- The number of nested **FOR / NEXT, WHILE / WEND** loops, block **IF / THEN / ELSE / ENDIF** and **GOSUB** subroutines is limited.

GOTO line / GOTO `label

Program mode only. Program execution continues by jumping to ***line*** or ***`label***.

INCLUDE path

Program or Direct mode. Includes ACS Basic statements from a SD card file specified by ***path***. The full ***path*** to the file must be specified and must not start with a leading backslash.

- Statements in the file without line numbers are immediately executed.
- Statements with line numbers are entered as if they were typed in – adding new or replacing existing numbered lines.
- Note that **INCLUDE** must be the ONLY statement on a line.
- If not present, the .BAS file extension on the filename at the end of the path is assumed.

IF test THEN line/`label/statement {ELSE line2/`label2/statement2}

Program mode only. Conditional execution jump. The expression ***test*** is evaluated, and if non-zero, program execution continues at ***line*** or ***`label*** or the single ***statement*** is executed. If the optional **ELSE** clause is present and the ***test*** expression evaluates to zero program execution continues at ***line2*** or `label2 or the single ***statement2*** is executed.

Some **IF** statement examples:

```
10 IF A=0 THEN 100
20 IF A=1 THEN GOTO 200
30 IF A=0 THEN PRINT “A was zero” ELSE 100
40 IF A=1 THEN PRINT “A was zero” ELSE PRINT “A non-zero”
```

Multiple conditions can be tested at the same time by combining two or more ***test*** expressions with the logical **AND**, **OR** operators:

```
20 IF (A=1) AND (B=2) THEN PRINT "Both A and B are correct"
30 IF (A=1) OR (B=2) THEN PRINT "Either A or B is correct" ELSE PRINT “Neither A or B”
```

IF test THEN

statements

{ELSE

statements}

ENDIF

Program mode only. Conditional execution block jump. There must be no statements on the same line following the **THEN**, **ELSE** and **ENDIF** keywords. The expression ***test*** is evaluated, and if non-zero, program execution continues with the following ***statements***. If the test expression evaluates to zero, program execution continues at the statements following the **ENDIF**. If the optional **ELSE** clause is present and the ***test*** expression evaluates to zero program execution continues at the ***statements*** following the **ELSE** up until the **ENDIF** is executed.

- As the conditional block **IF / THEN / ENDIF** uses the control stack to execute you should not jump out of or into either code block of statements. To leave the block **IF** statement **GOTO** the **ENDIF** statement.
- A **BREAK** statement can be used to exit a **FOR / NEXT** or **WHILE / WEND** loop from within a block **IF** statement.
- A **CONTINUE** statement can be used to continue a **FOR / NEXT** or **WHILE / WEND** loop from within a block **IF** statement.

- A **RETURN** statement can be used to return from a subroutine from within a block **IF** statement.

An **IF** / **THEN** / **ENDIF** statement example – notice the use of extra spaces to indent the statement blocks to improve readability:

```
10 REM block if
20 a=0
30 IF a=1 THEN
40  PRINT "if condition line 1"
42  PRINT "if condition line 2"
44  PRINT "if condition line 3"
50 ELSE
60  PRINT "else condition line 1"
62  PRINT "else condition line 2"
64  PRINT "else condition line 3"
70 ENDIF
```

INPUT var{$}

Get value for variable from the serial port.

INPUT "prompt", var

Get value of **var**iable from the serial port with prompt. Prompt may be a constant string or you can use a string variable in the prompt by concatenating it to a string: **INPUT ""+A$, B$**

INPUT #N, var

Get value for **var**iable from file or internet socket handle #***N*** (0 → 9). Note that when an end of file occurs, the **var**iable will have its last value. Test the **@FEOF[#N]** system variable to detect this condition when inputting from a file.

{LET} var{$}=expr{$} (default statement)

Program or Direct mode. Sets **var**iable = **expr**ession (This is the default statement, so the **LET** keyword is not required). An attempt to assign a string value to a numeric variable or a numeric value to a string variable will generate a “Type Error”. Some examples:

```
LET a0 = 240
100 Z9$ = “Test”
@TIMER[0] = 240
```

LIF test THEN statement{: statement …}

Program mode only. **L**ong **IF** (all statements to end of line). The expression ***test*** is evaluated, and if non-zero, all statements to the end of the current program line are executed.

```
20 LIF @CLOSURE[24]=1 THEN PRINT “25 closed”:GOSUB 100:@CLOSURE[24]=0
30 GOTO 20
```

Multiple conditions can be tested at the same time by combining two or more ***test*** expressions with the logical **AND**, **OR** operators:

```
20 LIF (A=0) AND (@CLOSURE[24]=1) THEN PRINT “25 closed”:GOSUB 100:@CLOSURE[24]=0
30 GOTO 20
```

LIST {start{, end}} … LIST {start{-end}}

Direct mode only. List program lines to the serial port. May also specify a starting and ending line number to limit the range of lines that are displayed. A double escape sequence will stop the portion of the file that the CFSound not already queued for output.

LIST #N {start{, end}} … LIST #N {start{-end}}

Direct mode only. List program lines to open file #***N*** (0 → 9). May also specify a starting and ending line number to limit the range of lines that are displayed. A double escape sequence will stop the portion of the file that the CFSound not already written.

LOAD path

Program or Direct mode. Load an ACS Basic program from a SD card file specified by ***path***. The full ***path*** to the program file must be specified and must not start with a leading backslash.

- When **LOAD** is used within a program, execution continues with the first line of the newly loaded program. In this case, the user variables are not cleared. This provides a means of chaining to a new program, and passing information to it.
- When used in a program note that **LOAD** must be the last statement on a line. If not present, the .BAS file extension on the filename at the end of the path is assumed.

```
load program1
Ready
list
10 PRINT "Program 1 A=",a
20 a=a+1
30 LOAD program2
Ready
load program2
Ready
list
10 PRINT "Program 2 A=",a:a=a+1:LOAD program1
Ready
run
Program 2 A= 0
Program 1 A= 1
Program 2 A= 2
Program 1 A= 3
 ESC at line 30
Ready
```

MD path

Direct mode only, requires a SD card. Makes a new directory on the SD card. ***Path*** must be a complete path for the new directory, and it must not already exist. ***Path*** may be a constant string or you can use a string variable as the ***path*** by concatenating it to such a string: ***MD*** ""+***PATH$***.

MEMORY

Displays the currently available program memory, resource memory and SD card memory if a SD card is present.

```
memory
SD card bytes free:     7,929,724,928
Resource bytes free:       23,938,413
Program bytes free:         8,385,960
Ready
```

NEW

Direct mode only. Erases all program statements, clears all variable values and closes all open files.

ON expr, GOSUB line0, line1, line2, … ,lineN

Program mode only. Case statement dispatching via subroutines. The value of ***expr*** is evaluated, and a subroutine call is performed to the ***line0*** statement if it evaluates to zero, ***line1*** if one, etc.

- If the value of ***expr*** is negative or greater than the number of line numbers present, execution continues with the next statement.
- A line number of zero in the list also causes execution to continue with the next statement if the expression evaluates to that position.
- Upon return from the **GOSUB** execution continues with the next statement.

```
5 REM ONGOSUB Demo
10 a=0
20 ON a,GOSUB 100,200,300,400
30 GOTO 20
100 PRINT "1",
105 a=a+1
110 RETURN
200 PRINT "2",
205 a=a+1
210 RETURN
300 PRINT "3",
305 a=a+1
310 RETURN
400 PRINT "4",
405 a=a+1
410 RETURN
Ready
run
1234
1234
1234
1234
1234
1 ESC at line 105
Ready
```

ON expr, GOTO line0, line1, line2, … , lineN

Program mode only. Case statement dispatching via jumps. The value of ***expr*** is evaluated, and a jump is performed to the ***line0*** statement if zero, ***line1*** if one, etc.

- If the value of ***expr*** is negative or greater than the number of line numbers present, execution continues with the next statement.
- A line number of zero in the list also causes execution to continue with the next statement if the expression evaluates to that position. Labels can also be used in place of non-zero line numbers.

```
5 REM ON GOTO DEMO
10 a=0
20 ON a,GOTO 100,200,300,400
30 GOTO 10
100 PRINT "1",
105 a=a+1
110 GOTO 20
200 PRINT "2",
205 a=a+1
210 GOTO 20
```

```
300 PRINT "3",
305 a=a+1
310 GOTO 20
400 PRINT "4",
405 a=a+1
410 GOTO 20
Ready
run
1234
1234
1234
1234 ESC at line 20
Ready
```

ONERROR GOTO line

Program mode only. Provides one-shot error handling. Upon any error, statement execution starts at line, and the **ERR( )** function has the value of the error number and the **ERR$( )** function has the string version of the error number. The **ONERROR** condition is then cleared so that subsequent errors result in program termination. The **ONERROR** can be disabled by specifying a ***line*** number of zero.

```
10 ONERROR GOTO 100
20 REM error follows
30 a=10/0
40 STOP
100 PRINT "Error #",ERR()," - ",ERR$()
Ready
run
Error # 6 - Divide by zero error in line 30
Ready
```

A common use of **ONERROR** statement is to allow execution of a command that might fail without causing the program to stop execution. For example if you want to delete a file with the **DEL** command, if the file didn't exist the **DEL** command would produce an error and the program would stop. By setting up an **ONERROR** handler to bracket the **DEL** command the program will continue execution if the file to be deleted did or did not exist:

```
170 ONERROR GOTO 180 : DEL "WAVLIST.TXT" : ONERROR GOTO 0
180 REM execution continues here even if WAVLIST.TXT didn't exist
```

The **ONERROR** statement can also be used to perform error logging. There is an example of error logging in the **ACS Basic Examples** section below.

ONEVENT @systemvar, GOSUB line

Program mode only. Provides semi-asynchronous event handling via subroutines. Certain ACS Basic system variables can trigger events. The **ONEVENT** statement allows the event to be associated with the execution of a subroutine. When the event occurs, after execution of any current statement that does not transfer control, control is transferred to the subroutine starting at ***line***. *While in the event subroutine, only higher priority events will be recognized until after the **RETURN** statement is executed.* An event handler can be disabled by specifying a ***line*** number of zero. Executing the ONEVENT statement clears the associated event in preparation for the subsequent event handling.

The following system variables can cause events and are listed in order of ***decreasing*** priority:

@TIMER[x]	The event occurs one time whenever the timer counts down to zero. System variable **@TIMER[0]** is the highest priority, followed by **@TIMER[1]**, … then **@TIMER[9]**. $0 \le x \le 9$
@CLOSURE(x)	The event occurs whenever a contact I/O contact has closed. $0 \le x \le 56$
@OPENING(x)	The event occurs whenever a contact has opened. $0 \le x \le 56$
@FEOF[#N]	The event occurs after **INPUT #N**, **FINPUT #N** or **FREAD #N** reaches the end of file #N (0 → 9)
@SECOND	The event occurs once per second.
@MINUTE	The event occurs once per minute.
@HOUR	The event occurs once per hour.
@DOW	The event occurs once per day at midnight.
@DATE	The event occurs once per day at midnight.
@MONTH	The event occurs once per month at midnight of day 1.
@YEAR	The event occurs once per year.
@SOUND$	The event occurs after the last queued **@SOUND$** sound has finished playing.
@MSG$	The event occurs after receipt of a serial character stream delineated by the **@SOM** and **@EOM** characters.
@EOT	The event occurs upon complete transmission of a serial character stream of one or more characters when both the output buffer and UART are empty.
@TOUCH.KEYPAD	The event triggers upon a change in the display touch keypad status.
@TOUCH.EVENT	The event triggers upon user touch interaction with the display touchscreen.
@SCREEN.OBJ.EVENT	The event triggers upon user touch interaction with the screen object.
@SCREEN.EVENT	The event triggers as each @SCREEN is displayed or hidden.

Each ACS Basic implementation may have additional system variables that may cause events. Please consult each product's manual for details.

Here is a short program that outputs the current time, once per second, on the serial port. Note that the program's idle loop, which it executes while waiting for the second event to occur, consists of a single **GOTO** statement jumping to itself:

```
5 REM print the time once per second
10 ONEVENT @SECOND,GOSUB 100
20 GOTO 20
100 PRINT CHR$(13);
105 PRINT FMT$("%2d",@HOUR);
110 PRINT ":";
115 PRINT FMT$("%02d",@MINUTE);
120 PRINT ":";
125 PRINT FMT$("%02d",@SECOND);
130 RETURN
Ready
run
14:47:15 ESC at line 30
Ready
```

OPEN #N, "path", "options"

Open filename ***path*** as file #***N*** (0 → 9) for subsequent access via **DIR #N**, **INPUT #N, FINPUT #N, PRINT #N** or **FPRINT #N** statements. The ***options*** string characters are:

"options"	**Description**
"r"	opens file for reading, if ***path*** does not exist an error is generated
"w"	opens file for writing, if ***path*** exists its contents are destroyed
"r+"	opens file for read and write, the ***path*** must exist
"w+"	opens an empty file for read and write, if ***path*** exists its contents are destroyed
"a+"	opens file for reading and appending (seek to end of file after open)
"b"	opens file in binary mode, no translations
"t"	opens file in text mode (default), CR/LF pairs are translated to LF on input and LF translated to CR/LF pairs on output.

ORDER line

Program mode only. This statement positions the read data pointer to statement ***line*** number. The statement at ***line*** must be a series of one or more **DATA** statement.

PLAY file

Plays the sound ***file*** and waits until it completes. Program execution then continues with the next statement. If the ***file*** is not a valid .WAV file of the correct format, sample rate and sample size for the CFSound-IV an "Invalid .WAV File Error" is generated.

- ***File*** may be a constant string or you can use a string variable as the ***file*** by concatenating it to such a string: ***PLAY*** ""***+PATH$***. While the sound file is playing
- Events can occur during the **PLAY** statement, but any defined **ONEVENT** handlers will not be executed until the sound has finished playing.
- To play sounds while continuing program execution use the **@SOUND$** system variable.

PRINT expr{$}{, expr{$} ...}{,} … PRINT expr{$} {; expr{$} …}{;}

Prints one or more expression(s) to the serial port.

- If the expressions are separated by a comma(',') then a space is output in between.
- If the expressions are separated by a semicolon (';') then no space is output.
- If the statement ends in a comma or semicolon no Carriage Return / Line Feed pair is appended to the printed expressions allowing multiple print statements to display on the same line.

When **@ANSI.ENABLE=1**, the **PRINT** statement is also shown on the CFSound virtual display as drawn ANSI text. See the ANSI Mode for information on how the text is rendered on the CFSound.

PRINT #N, expr{$}{, expr{$} ...} … PRINT #N, expr{$}{; expr{$} …}

Prints one or more expressions to a previously opened file or internet streaming handle #**N** (0 → 9). The same formatting conditions as the **PRINT** statement above apply.

PRINT USING fmt$ {, expr{$} {, expr{$} … , expr{$}}}{;}

Prints zero or more formatted numeric or string **expr**essions to the serial port.

- The individual expressions are formatted with the specifications in the **fmt$** string using the same format specification as the **FMT$( )** function above.
- If the statement ends in a semicolon no Carriage Return / Line Feed pair is appended to the printed expressions allowing multiple print statements to display on the same line.

When **@ANSI.ENABLE=1**, the **PRINT** statement is also shown on the CFSound virtual display as drawn ANSI text. See the ANSI Mode for information on how the text is rendered on the CFSound.

PRINT #N, USING fmt$ {, expr{$} {, expr{$} … , expr{$}}}{;}

Prints zero or more formatted numeric or string **expr**essions to a previously opened file or internet streaming handle #**N**(0→ 9).

- The individual expressions are formatted with the specifications in the **fmt$** string using the same format specification as the **FMT$( )** function above.
- If the statement ends in a semicolon no Carriage Return / Line Feed pair is appended to the printed expressions allowing multiple print statements to display on the same line.

```
10 REM show time
15 ONEVENT @SECOND,GOSUB 100
20 GOTO 20
100 PRINT USING "%c%2d:%02d:%02d",13,@HOUR,@MINUTE,@SECOND;
105 RETURN
Ready
run
13:28:52 <<< ESC at line 20 >>>
Ready
```

READ var{$}{, var{$} ... , var{$}}

Program mode only. Reads data from **DATA** program statements into **var**iables. You *MUST* issue an **ORDER** statement targeting a line containing a valid **DATA** statement before using **READ**.

RETURN

Program mode only. Return from a subroutine invoked via a **GOSUB** statement.

- You can **RETURN** from within a **FOR / NEXT**, **WHILE / WEND** or block **IF** statement inside of a subroutine.
- A return without a prior **GOSUB** will generate a “Stack Error”.

REM

Comment... the remainder of line is ignored. Used to document the operation of the program.

REN oldfile newfile

Renames oldfile to newfile. ***Oldfile*** and ***newfile*** may be constant strings or you can use string variables as the ***files*** by concatenating them to string constants: **REN** ""**+OLD$,** ""**+NEW$**. In Direct mode the quotes are not required.

RESQ {start{-end}{,new}{,incr}}

Direct mode only. Re-sequences the program line numbers from ***start*** through ***end*** beginning with the value of ***new*** advancing by ***incr***. The default value of ***start*** is the first line of the program, the default for ***end*** is the last line of the program, the default for ***new*** is 10 and the default for ***incr*** is 5.

- The program is renumbered with all embedded references to the new line numbers corrected. It is displayed and written to a file with the same name as the original program with the extension **.RSQ**.
- If there are syntax errors in the program, or references to non-existent line numbers, the **RESQ** will error and stop. The original program should be **SAVE**d before attempting to re-sequence it.
- No checks are made to avoid overlapping line numbers and the generated **.RSQ** file should be loaded, viewed and run before saving it over the original program file.

```
list
10 ON N,GOTO 100,150,200
20 GOSUB 250
30 GOTO 30
100 REM
150 REM
200 STOP
250 RETURN
Ready
resq
Writing resequenced program to:test2.RSQ

10 ON N,GOTO 25,30,35
15 GOSUB 40
20 GOTO 20
25 REM
30 REM
35 STOP
40 RETURN
```

RUN {line} … RUN path

Direct mode only. Executes the program starting at the lowest or optional ***line*** number. If path is present, **LOADs** and **RUNs** a file directly at the lowest line number.

- If not present, the .BAS file extension on the filename at the end of the path is added.

SAVE {path}

Direct mode only. Saves the current program to a disk file on the SD card with the filename specified in ***path***, or to the filename in the previous **LOAD, SAVE** or **RUN** command if not specified.

- If not present, the .BAS file extension on the filename at the end of the path is added.
- If the provided name doesn't match the previous LOAD, SAVE or RUN filename, and the file already exists an overwrite warning message is displayed requiring approval.

SEARCH string {filename]

Direct mode only. Performs a case insensitive search for the occurrence of the **string** displaying where it appears. The search **string** can be a program statement keyword or contain '*' and '?' wildcard characters to match multiple variations of the search string.

- If the optional **filename** is not present the currently loaded program is the search target for the occurrence of the search **string** displaying the lines where it appears.
- If the optional **filename** is present then that file becomes the search target and the file's lines are searched for the occurrence of the search string. The optional **filename** can contain '*' and '?' wildcard characters to search multiple files.

SIGNAL @systemvar

Signal an event associated with System variable.

SORT var{$}

Sorts an array of strings or integers in ascending order. Here's an example of sorting an integer array:

```
10 REM sorting integers
15 CONST size = 20
20 DIM a[size]
30 FOR n = 0 TO size-1 : a[n] = RND(1000) : NEXT n
40 PRINT "before sort:" : FOR n = 0 TO size-1 : PRINT a[n];" "; : NEXT n
50 SORT a : PRINT ""
60 PRINT "after soft:" : FOR n = 0 TO size-1 : PRINT a[n];" "; : NEXT n
70 PRINT ""
Ready
run
before sort:
540 808 884 101 604 388 201 802 712 350 948 793 615 305 477 455 563 526 136 481
after soft:
101 136 201 305 350 388 455 477 481 526 540 563 604 615 712 793 802 808 884 948
Ready
```

And here's an example of sorting a string array:

```
10 REM sorting strings
15 CONST size = 10
20 DIM a$[size]
30 FOR n = 0 TO size-1 : FOR i = 0 TO 5 : a$[n] = a$[n] + CHR$(RND(25)+65) : NEXT i : NEXT n
40 PRINT "before sort:" : FOR n = 0 TO size-1 : PRINT a$[n];" "; : NEXT n
50 SORT a$ : PRINT ""
60 PRINT "after soft:" : FOR n = 0 TO size-1 : PRINT a$[n];" "; : NEXT n
70 PRINT ""
Ready
run
before sort:
AWRLHC APIAER XMDYBJ UDEMUI EJODGW HWWYJC VFRHND NEVMPH UJIPJG TKGKRK
after soft:
APIAER AWRLHC EJODGW HWWYJC NEVMPH TKGKRK UDEMUI UJIPJG VFRHND XMDYBJ
Ready
```

SMTP.SERVER "name", "ipaddress"{, port{, "usernameb64", "passwordb64"}}

Program mode only. Prepares the SMTP network stack with parameters for a subsequent SMTP.SEND command.

- The first two parameters are required and specify the mail server name, and the mail server's IPv4 address.
- The third parameter specifies the TCP/IP port which defaults to 25 if not specified.

- The last two parameters are optional and are used to perform an authenticated login in response to an AUTH LOGIN status from the mail server. The usernameb64 is the response sent to the Username: prompt encoded as a Base64 string and the passwordb64 is the response sent the Password: prompt encoded as a Base64 string. No other authentication mechanism is supported.

Specify a connection to an intranet mail relay at IPv4 address 192.168.1.100 on port 25:

20 SMTP.SERVER "192.168.1.100", "192.168.1.100"

Specify a connection to mail server "mail.domain.com" at IPv4 address 216.119.100.100 on port 2525 with **usernameb64** of "sender@domain.com" base64 encoded as " c2VuZGVyQGRvbWFpbi5jb20=" and **passwordb64** of "1234" base64 encoded as " MTIzNA==". An online tool such as http://www.base64encode.org used to encode the username and password:

20 SMTP.SERVER "mail.domain.com","216.119.100.100",2525," c2VuZGVyQGRvbWFpbi5jb20="," MTIzNA=="

SMTP.SEND "from", "to", "cc", "subject", "message"

Sends a text "message" via the mail server previously specified by the SMTP.SERVER command "from", "to", "cc" with "subject".

In this example a single string message is sent without login using an intranet mail relay:

```
10 REM smtp test
15 ONEVENT @SMTP.EVENT, GOSUB 100
20 SMTP.SERVER "192.168.1.100", "192.168.1.100"
25 SMTP.SEND "cfsound4@acscontrol.com", "support@acscontrol.com", "", "test", "this is a simple
text message"
30 GOTO 30
100 REM SMTP Event Handler
105 ON @SMTP.EVENT,GOTO 115,120,125,130,135,140,145,150,155,160,165
110 PRINT "unknown event" : RETURN
115 PRINT "SMTP OK: " + @SMTP.MESSAGE$ : RETURN
120 PRINT "SMTP ABORTED: " + @SMTP.MESSAGE$ : STOP
125 PRINT "SMTP CONNECT: " + @SMTP.MESSAGE$ : STOP
130 PRINT "SMTP HELO: " + @SMTP.MESSAGE$ : STOP
135 PRINT "SMTP AUTH: " + @SMTP.MESSAGE$ : STOP
140 PRINT "SMTP FROM: " + @SMTP.MESSAGE$ : STOP
145 PRINT "SMTP TO: " + @SMTP.MESSAGE$ : STOP
150 PRINT "SMTP CC: " + @SMTP.MESSAGE$ : STOP
155 PRINT "SMTP DATA: " + @SMTP.MESSAGE$ : STOP
160 PRINT "SMTP QUEUE: " + @SMTP.MESSAGE$ : STOP
165 PRINT "SMTP SUCCESSFUL: " + @SMTP.MESSAGE$ : STOP
Ready
run
SMTP OK: 250-acsservr Hello [192.168.1.200]
SMTP OK: 250-TURN
SMTP OK: 250-ATRN
SMTP OK: 250-SIZE 2097152
SMTP OK: 250-ETRN
SMTP OK: 250-PIPELINING
SMTP OK: 250-DSN
SMTP OK: 250-ENHANCEDSTATUSCODES
SMTP OK: 250-8bitmime
SMTP OK: 250-BINARYMIME
SMTP OK: 250-CHUNKING
SMTP OK: 250-VRFY
SMTP OK: 250 OK
SMTP SUCCESSFUL: 221 2.0.0 acsservr Service closing transmission channel

STOP in line 165
Ready
```

Here's the received e-mail with headers:

```
Return-Path: <cfsound4@acscontrol.com>
Received: from acsservr (pool-96-252-xxx-xxx.tampfl.fios.verizon.net [96.252.xxx.xxx]) by
maila20.webcontrolcenter.com with SMTP;
   Wed, 12 Mar 2014 08:24:37 -0700
Received: from 192.168.1.100 ([192.168.1.200]) by acsservr with Microsoft SMTPSVC(5.0.2195.7381);
                Wed, 12 Mar 2014 10:24:36 -0500
From: cfsound4@acscontrol.com
To: support@acscontrol.com
Subject: test
Return-Path: cfsound4@acscontrol.com
Message-ID: <ACSSERVRhzsDYlYhiIA000002cf@acsservr>
X-OriginalArrivalTime: 12 Mar 2014 15:24:36.0421 (UTC) FILETIME=[2D634F50:01CF3E07]
Date: 12 Mar 2014 10:24:36 -0500
X-SmarterMail-TotalSpamWeight: 0 (Authenticated)

this is a simple text message
```

SMTP.SEND #N, “from”, “to”, “cc”, “subject” {, “header”}

Sends the contents of a previously opened file #N via the mail server previously specified by the SMTP.SERVER command “from”, “to”, “cc” with “subject” and optional “header” that can specify MIME content headers to allow HTML e-mails to be sent from a file.

In this example a HTML file is sent via mail server “mail.domain.com” at the specified IPv4 address and port using an authenticated login and the requisite MIME content header:

```
10 REM smtp test
15 ONEVENT @SMTP.EVENT, GOSUB 100
20 SMTP.SERVER "mail.domain.com",”216.119.100.100”,2525," c2VuZGVyQGRvbWFpbi5jb20="," MTIzNA=="
25 OPEN #0,"SampleEmail.html","rb"
30 Header$ = "MIME-Version 1.0" + CHR$(13)+CHR$(10) + "Content-Type: text/html; charset=" +
CHR$(34) + "ISO-8859-1" + CHR$(34) + "; "
35 SMTP.SEND #0, "cfsound4@domain.com", "support@acscontrol.com", "", "test", Header$
40 GOTO 40
100 REM SMTP Event Handler
105 ON @SMTP.EVENT,GOTO 115,120,125,130,135,140,145,150,155,160,165
110 PRINT "unknown event" : RETURN
115 PRINT "SMTP OK: " + @SMTP.MESSAGE$ : RETURN
120 PRINT "SMTP ABORTED: " + @SMTP.MESSAGE$ : STOP
125 PRINT "SMTP CONNECT: " + @SMTP.MESSAGE$ : STOP
130 PRINT "SMTP HELO: " + @SMTP.MESSAGE$ : STOP
135 PRINT "SMTP AUTH: " + @SMTP.MESSAGE$ : STOP
140 PRINT "SMTP FROM: " + @SMTP.MESSAGE$ : STOP
145 PRINT "SMTP TO: " + @SMTP.MESSAGE$ : STOP
150 PRINT "SMTP CC: " + @SMTP.MESSAGE$ : STOP
155 PRINT "SMTP DATA: " + @SMTP.MESSAGE$ : STOP
160 PRINT "SMTP QUEUE: " + @SMTP.MESSAGE$ : STOP
165 PRINT "SMTP SUCCESSFUL: " + @SMTP.MESSAGE$ : STOP
Ready
run
SMTP OK: 250-maila20.webcontrolcenter.com Hello [96.252.xxx.xxx]
SMTP OK: 250-SIZE 31457280
SMTP OK: 250-AUTH LOGIN CRAM-MD5
SMTP OK: 250 OK
SMTP SUCCESSFUL: 221 Service closing transmission channel

STOP in line 165
Ready
```

Here is the contents of file SampleEmail.html – notice that the file is opened for reading in binary mode:

```
<html>
<body>
This is a test HTML e-mail message sent from a file
<p><p>
<a href="http://www.cfsound.com">http://www.cfsound.com</a>
```

```
</body>
</html>
```

And here's the received e-mail (with headers):

```
Return-Path: <cfsound4@acscontrol.com>
Received: from mail.domain.com (pool-xxx-xxx-xxx-xxx.tampfl.fios.verizon.net
[xxx.xxx.xxx.xxx]) by maila20.webcontrolcenter.com with SMTP;
   Tue, 11 Mar 2014 07:27:16 -0700
From: cfsound4@domain.com
To: support@acscontrol.com
Subject: test
MIME-Version: 1.0
Content-Type: text/html; charset="ISO-8859-1";
Content-Transfer-Encoding: 8bit;
Message-ID: <58e95feaed60455b987e168b4a279b33@com>
X-SmarterMail-TotalSpamWeight: 0 (Authenticated)
```

This is a test HTML e-mail message sent from a file
http://www.cfsound.com

SOCKET.ASYNC.CONNECT #N, "ip:port", connect( ), send( ), recv( )

Initiates an outgoing asynchronous network socket connection as file #N using the string representation of the IPv4 IP address and port number. If the socket opens without error execution continues with the following statement. The status of the connection, send, receive and disconnect process is returned via the **@SOCKET.EVENT[#N]** system variable.

- The **connect( )** user function is called when a connection is established.
- The **send( )** user function is called to send data to the connected device using PRINT #N or FPRINT #N statement(s). Return zero to terminate the connection, one to proceed to the **recv( )** function and two to be called again to send more data.
- The **recv( )** user function is then called to receive data from the connected device using INPUT #N or FINPUT #N statements(s). Return zero to terminate the connection, one to return to the **send( )** function and two to be called again to receive more data.

See the Socket Programming section below for more information and sample programs.

SOCKET.ASYNC.LISTEN #N, ":port", connect( ), recv( ), send( )

Initiates an incoming asynchronous network socket reception as file #N using the string representation of the IPv4 port number. If the socket opens without error execution continues with the following statement. The status of the connection, receive, send and disconnect process is returned via the **@SOCKET.EVENT[#N]** system variable.

- The **connect( )** user function is called when a connection is established.
- The **recv( )** user function is then called to receive data from the connected device using INPUT #N or FINPUT #N statements(s). Return zero to terminate the connection, one to return to the **send( )** function and two to be called again to receive more data.
- The **send( )** user function is called to send data to the connected device using PRINT #N or FPRINT #N statement(s). Return zero to terminate the connection, one to proceed to the **recv( )** function and two to be called again to send more data.

See the Socket Programming section below for more information and sample programs.

STOP

Program mode only. Terminates the program and issues a **STOP** message. Closes all open files.

```
10 a=a+1
20 STOP
Ready
run
STOP in line 20
Ready
```

TYPE path

Displays the contents of a SD card filename named ***path*** as ASCII characters on the serial port. ***Path*** may be a constant string or you can use a string variable as the ***path*** by concatenating it to such a string: ***TYPE*** ""+***PATH$***. In Direct mode the quotes are not required.

A double escape sequence will stop the portion of the file that the CFSound not already queued.

VARS

Displays a table of the name, type and values of the variables that have been defined or created by use.

WAIT @systemvar

Execution pauses at this statement until the associated system variable has been signaled.

Note that all statements on the same line before the WAIT are executed continuously while waiting.

In this example, program execution would pause at line 110 until all of the queued sounds had finished playing:

```
10 @SOUND$="one.wav"
20 @SOUND$="two.wav"
30 @SOUND$="three.wav"
40 @SOUND$="four.wav"
50 @SOUND$="five.wav"
60 @SOUND$="six.wav"
70 @SOUND$="seven.wav"
80 @SOUND$="eight.wav"
90 @SOUND$="nine.wav"
100 @SOUND$="ten.wav"
110 WAIT @SOUND$
```

In this example, program execution would pause at line 40 until all of the queued serial data had finished sending:

```
10 REM test @EOT
20 FOR I=1 TO 10:PRINT "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ": NEXT I
40 WAIT @EOT
50 PRINT "EOT"
Ready
run
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
EOT
Ready
```

In this ***incorrect example***, program execution would lock forever on line 20 since all statements on the same line before the WAIT are executed continuously while waiting. Since these statements reload the timer that the WAIT is waiting on, the program will never execute past this line:

```
5 REM Wrong use of the WAIT statement
10 PRINT "start timer[]:wait timer[]"
20 @TIMER[0]=50:WAIT @TIMER[0]
30 PRINT "done"
Ready
run
start timer():wait timer()
 ESC at line 20
Ready
```

WHILE test : statements : WEND

Program mode only. Conditional execution code block loop. The expression ***test*** is evaluated, and if non-zero, program execution continues with the following ***statements***. If the test expression evaluates to zero, program execution continues at the statements following the **WEND**.

- **WHILE / WEND** blocks can be nested and don't have to be the only statements on the line.
- **WHILE / WEND** blocks can be exited from within without the test expression evaluating to zero using the **BREAK** statement.
- **WHILE / WEND** blocks can be continued from within without executing all of the statements using the **CONTINUE** statement.
- As the **WHILE / WEND** code block loop uses the control stack to execute you should not jump out of or into the code block loop of statements. To leave the loop, force the **test** expression to return zero or use the **BREAK** statement.

Here are some examples:

```
10 REM while/wend test
15 WHILE a < 10 : PRINT a : a = a + 1 : WEND
Ready
run
0
1
2
3
4
5
6
7
8
9
Ready
```

```
10 REM while/wend test2
15 WHILE a < 10
20  PRINT a
25  a = a + 1
30 WEND
Ready
run
```

```
0
1
2
3
4
5
6
7
8
9
Ready
```

```
10 REM while/wend test2
15 WHILE a < 10
20  WHILE a < 5
22   a = a + 1
23  WEND
25  a = a + 1
30  PRINT a
35 WEND
Ready
run
6
7
8
9
10
Ready
```

Display Statements

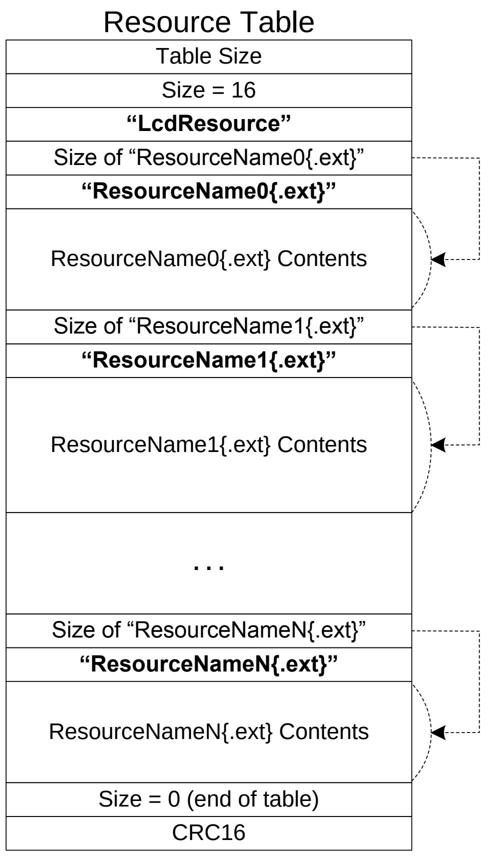
The following statements operate directly or indirectly on the Color LCD display. There are statements to manage resources such as fonts and images, perform drawing operations and configure and manage a higher-level screen setup.

RESOURCES

The ACS Color LCD 320x240 uses 'resources' to provide fonts, bitmaps and sounds in this mode. These resources are loaded into a 'table' in RAM where they can be rapidly accessed by the different commands that require them.

This resource table can be loaded from the on-board Serial Flash or from the micro SD card upon reset, or dynamically from the micro SD card during operation. The table is searched from the beginning when resources are accessed, with newly added resources located at the front of the table.

Each entry in the table consists of the size of the resource, its name– including any extension, and then the resource contents. The entire Resource Table is protected by a cyclic redundancy check value of its contents. Dynamically added resources are added to the front of the Resource Table:


Resource Table
Table Size
Size = 16
"LcdResource"
Size of "ResourceName0{.ext}"
"ResourceName0{.ext}"
ResourceName0{.ext} Contents
Size of "ResourceName1{.ext}"
"ResourceName1{.ext}"
ResourceName1{.ext} Contents
. . .
Size of "ResourceNameN{.ext}"
"ResourceNameN{.ext}"
ResourceNameN{.ext} Contents
Size = 0 (end of table)
CRC16

RESOURCES.INIT

Initializes the Resource table, clearing all entries.

RESOURCES.LIST {pattern}

Prints a list of the current contents of the Resource table. Wildcards '*' and '?' may be used in an optional pattern to select which resources are listed.

RESOURCES.FLASHERASE

Erases the on-board serial EEProm.

RESOURCES.FLASHLOAD

Initializes then loads the Resource table from the on-board serial EEProm.

RESOURCES.FLASHSTORE

Saves the current contents of the Resource table to the on-board serial EEProm.

RESOURCES.LOAD filename{.bin}

Initializes then loads the Resource table from the binary file on the micro SD card. The binary resource file image can be generated by an ACS Windows utility that supports the concept of resource generation projects (.RSRCGEN XML project files). (***See the appendix Resource File Generation in the Color_320x240_LCD_Display_Terminal manual)***

RESOURCES.SAVE filename{.bin}

Saves the current contents of the Resource table as a binary file on the microSD card.

RESOURCES.ADD filename

Adds the contents of filename from the micro SD card to the beginning of the Resource table. Duplicate resources with the same name are not allowed.

RESOURCES.REMOVE resourceName

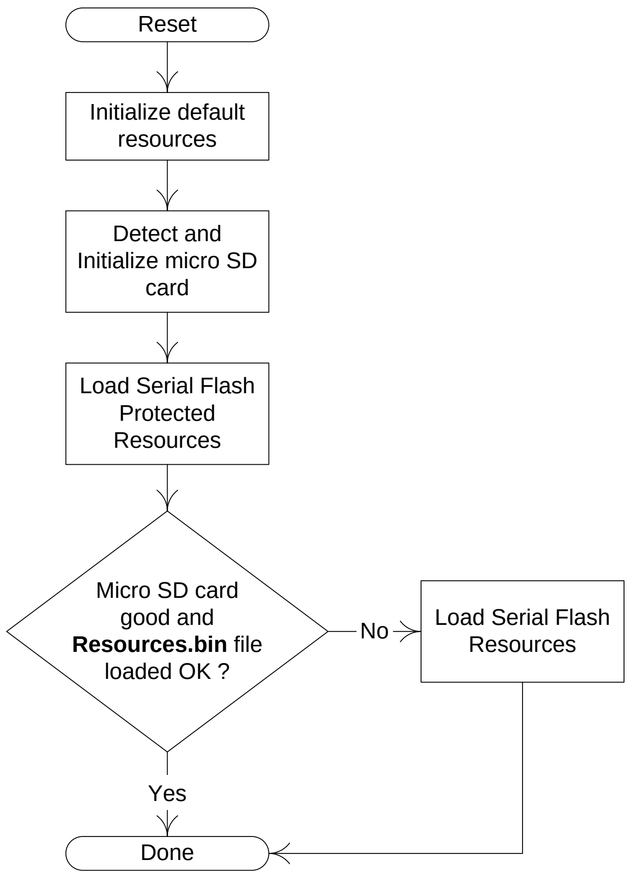
Removes the resource named resourceName from the Resource table.

RESOURCES.EXTRACT {pattern}

Extract the current contents of resources in the Resource table that match the pattern to the micro SD card. Wildcards '*' and '?' may be used in an optional pattern to select which resources are extracted. The user is prompted for existing file overwrite permission.

Initial Resource Table Load

When the ACS Color LCD 320x240 is powered-up or reset, it attempts to locate and load the resource table from several sources. This process is outlined in the following flowchart:

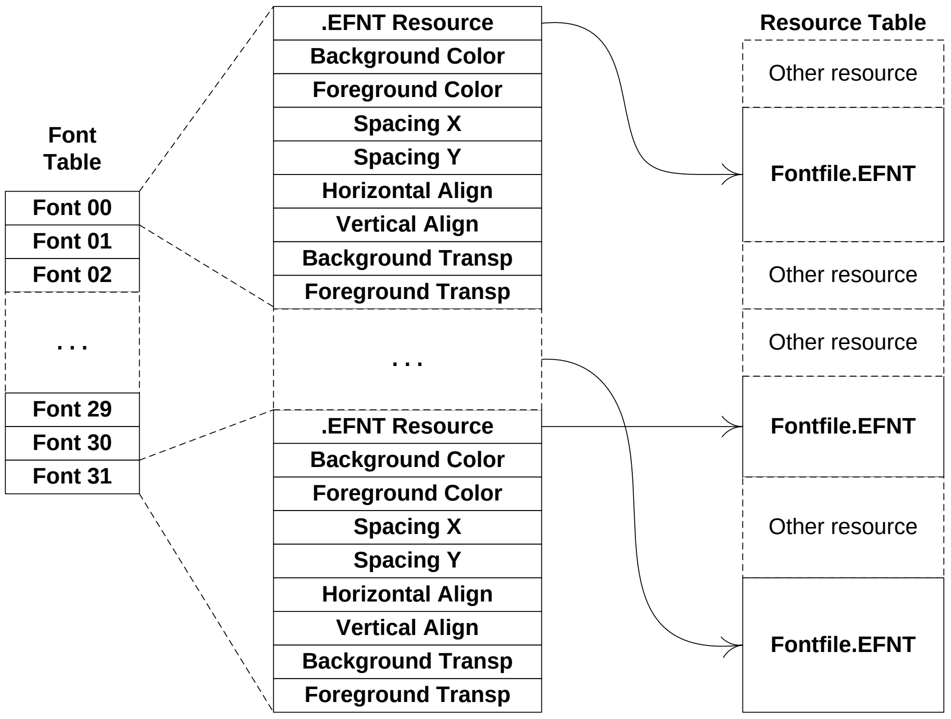

Reset
Initialize default resources
Detect and Initialize micro SD card
Load Serial Flash Protected Resources
Micro SD card good and **Resources.bin** file loaded OK ?
No
Load Serial Flash Resources
Yes
Done

1. The AcsDefaultFont.efnt (used for the diagnostics and keypad keys) and the AcsAnsiFont.efnt resources are loaded from internal flash memory.
2. The microSD card is detected and, if present, the file system is initialized.
3. A check is made for protected resources located at the top of the external Serial Flash memory and they are loaded if found.
4. If the microSD card was detected and initialized in step 2 attempt to load the binary resources file named: Resources.bin. If it loads and verifies then initial resource table loading is complete.
5. If step 4 fails, attempt to load and verify the resources located at the bottom of the external Serial Flash memory.

FONTS

The appearance of all text that is drawn on the display is controlled by fonts selected from a font table. The character images are extracted from previously constructed and loaded .EFNT resources. Each table entry holds the Font Name (.EFNT resource name), several attributes for horizontal and vertical alignment, spacing and background/foreground colors and transparency.

The table holds 32 entries. The table entry number is used as the selector of the font for drawing and display access.



The following commands are provided to manage the contents of the Font table.

FONTS.INIT

Initializes the Font table, setting all of the entries to defaults.

FONTS.LIST {#N} {start{, end}} … FONTS.LIST {start{-end}}

Prints a list of the current contents of the Font table, optionally to a previously opened file {#N}. May also specify a starting and ending font number to select which font numbers to list.

FONTS.LOAD filename{.fonts}

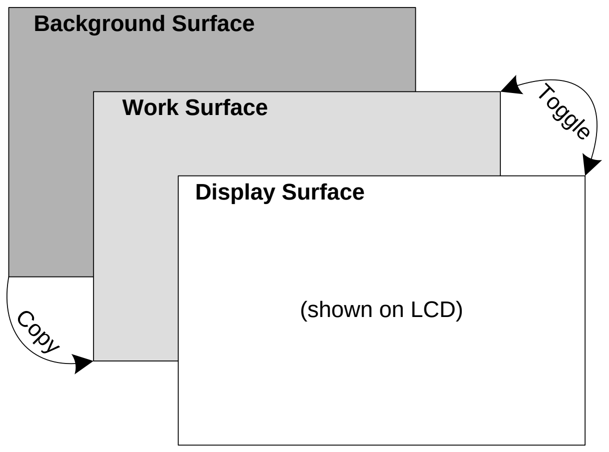
Loads the Font table from a JSON formatted file on the microSD card. (***See the appendix JSON File Formats in the Color 320x240 LCD Display Terminal User's Manual)***

FONTS.SAVE filename{.fonts}

Saves the current contents of the Font table to a JSON formatted file on the microSD card. (***See the appendix JSON File Formats in the Color 320x240 LCD Display Terminal User's Manual)***

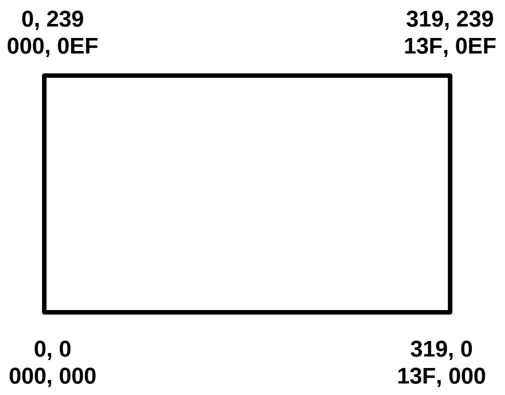
DRAWING

There are three drawing surfaces: Display, Work and Background. The Display surface is the one that is currently being shown on the LCD. The Work surface is where most drawing is done before being swapped with the Display surface. The Background surface provides a content area that can be seldom drawn, and copied to the Work surface before additional drawing is done on top.



All drawing takes place on the currently selected draw surface. Commands are provided to render pixels, lines, fills, boxes, circles, ellipses, text and images.

The X and Y coordinate for drawing start at **0, 0** at the lower left corner of the display:



Drawing is clipped at the coordinate boundaries of the display.

In order to avoid flickering as the display is updated the following sequence should be utilized:

7. Switch to the Background draw surface.
8. Draw any required static background content.
9. Switch to the Work surface.
10. Copy in the content from the Background surface to the Work surface and draw any dynamic content on top of it.
11. Toggle the Work and Display surfaces.
12. Repeat from step 4.

DRAW.INIT

Initializes the drawing system: clears the three drawing surfaces, restores the @ANSI system variables to their default values, sets the current drawing @SURFACE as DISPLAY.

DRAW.COPY sourceSurface, sX, sY, dX, dY, width, height

Copies the region located at sX,sY, of width and height from the specified source surface to the current drawing @SURFACE at dX, dY.

DRAW.TRANSLATE sourceSurface, sX, sY, dX, dY, width, height, mode

Translates the region located at sX,sY, of width and height from the source surface to the current drawing @SURFACE at dX, dY using mode.

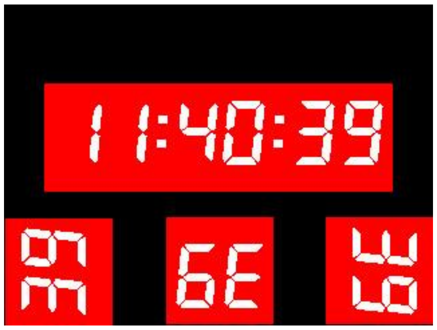
Where:

mode	Operation
0	No rotation
1	Rotate 90 CCW
2	Rotate 180 CCW
3	Rotate 270 CCW
+4	Flip horizontal
+8	Flip vertical

Here's a sample program that displays the time once per second and shows three copies of the seconds portion each rotated by 90 degrees:

```
10 REM demo draw.translate
15 @ANSI.ENABLE = 0:@ANSI.CURSOR = 0:@BACKLIGHT = 1
20 REM configure font[0]
25 @FONT.EFNT$[0]="Fnt-LiquidCrystal64.efnt":@FONT.FCOLOR[0]=RGB(255,255,255)
30 @FONT.BTRANSP[0] = 1 : @FONT.HALIGN[0] = 1 : @FONT.VALIGN[0] = 1
35 REM draw filled box on background
40 @SURFACE = 2 : DRAW.BOX.FILLED 30, 100, 290, 180, RGB(255,0,0) : @SURFACE = 1
45 REM init @SECOND event handler
50 ONEVENT @SECOND,GOSUB 1005
55 GOTO 55
1000 REM @SECOND event handler
1005 t$ = FMT$("%02d",@HOUR) + ":" + FMT$("%02d",@MINUTE) + ":" + FMT$("%02d",@SECOND)
1010 REM copy background to work, draw current time on work
1015 DRAW.COPY 2, 30, 100, 30, 100, 260, 150 : DRAW.TEXT 0, 160, 140, t$
1020 REM translate and rotate seconds on work
1025 DRAW.TRANSLATE 1, 210, 100, 0, 0, 80, 80, 1
1030 DRAW.TRANSLATE 1, 210, 100, 120, 0, 80, 80, 2
1035 DRAW.TRANSLATE 1, 210, 100, 240, 0, 80, 80, 3
1040 REM toggle work and display
1045 DRAW.TOGGLE
1050 RETURN
```

And the resulting screen display:



DRAW.TOGGLE

Toggles the DISPLAY and WORK drawing surfaces – WORK becomes DISPLAY and DISPLAY becomes WORK. This is useful for double-buffer or ping-pong drawing to minimize flickering.

DRAW.PIXEL x, y, color

Sets the display pixel at location x, y on the current drawing @SURFACE to color.

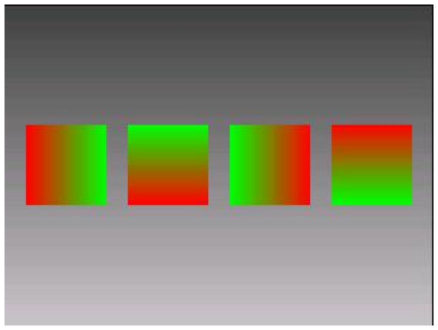
DRAW.FILL x, y, width, height, color {, endcolor, angle}

Fills the display starting at x, y for width and height with color. If endcolor and angle arguments are present fills as a gradient from color to endcolor at angle. Angle can be 0, 90, 180 or 270.

Here's a sample program that shows an overall gradient fill, then smaller fills at the four different angles:

```
10 REM DRAW.FILL demo
15 @ANSI.ENABLE = 0 : @BACKLIGHT = 1
20 DRAW.FILL 0, 0, 319, 239, RGB(192,192,192), RGB(64,64,64), 90
25 DRAW.FILL 16, 90, 60, 60, RGB(255,0,0),RGB(0,255,0),0
30 DRAW.FILL 92, 90, 60, 60, RGB(255,0,0),RGB(0,255,0), 90
35 DRAW.FILL 168, 90, 60, 60, RGB(255,0,0),RGB(0,255,0), 180
40 DRAW.FILL 244, 90, 60, 60, RGB(255,0,0),RGB(0,255,0), 270
50 GOTO 50
```

And the resulting screen display:



DRAW.LINE startX, startY, endX, endY, color

Draws a line of color from startX, startY to endX, endY.

DRAW.LINE.DASHED startX, startY, endX, endY, color, pattern, scale

Draws a dashed line of color from startX, startY to endX, endY. Each bit in the pattern is used scale times to draw the pixels in the line; a pattern of 21845 (hex 5555) (binary 0101010101010101) with a scale of 4 would draw 4 pixel dashes, 4 pixels apart.

DRAW.ARC x, y, width, height, startDegrees, endDegrees, color

Draws an arc centered at x, y of width and height through angle startDegrees to endDegrees using color.

DRAW.ARC.STYLED x, y, width, height, startDegrees, endDegrees, color, style

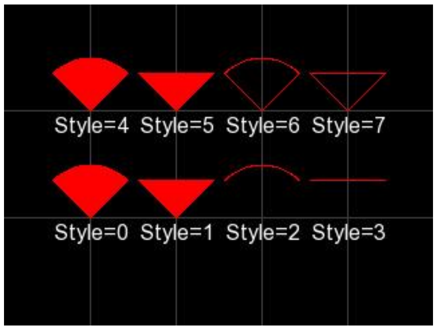
Draws an arc centered at x, y of width and height through angle startDegrees to endDegrees using color styled by the following style bits – zero degrees is right and angle advances counter-clockwise:
Where:

Style Bit	Name
1	Chord
2	No Fill
4	Edged

Here's a sample program that shows the results of combining the various style bits:

```
10 REM arc demo
20 @ANSI.ENABLE=0
30 REM draw grid
40 FOR x = 64 TO 256 STEP 64 : DRAW.LINE x,0,x,239,RGB(64,64,64) : NEXT x
50 FOR y = 80 TO 160 STEP 80 : DRAW.LINE 0,y,319,y,RGB(64,64,64) : NEXT y
60 REM init text font
70 @FONT.HALIGN[0]=1 : @FONT.VALIGN[0]=2 : @FONT.FCOLOR[0]=RGB(255,255,255)
80 REM draw eight styles of filled arcs
90 DRAW.ARC.FILLED 64,80,80,80,45,135,RGB(255,0,0),0
100 DRAW.TEXT 0,64,80,"Style=0"
110 DRAW.ARC.FILLED 128,80,80,80,45,135,RGB(255,0,0),1
120 DRAW.TEXT 0,128,80,"Style=1"
130 DRAW.ARC.FILLED 192,80,80,80,45,135,RGB(255,0,0),2
140 DRAW.TEXT 0,192,80,"Style=2"
150 DRAW.ARC.FILLED 256,80,80,80,45,135,RGB(255,0,0),3
160 DRAW.TEXT 0,256,80,"Style=3"
170 DRAW.ARC.FILLED 64,160,80,80,45,135,RGB(255,0,0),4
180 DRAW.TEXT 0,64,160,"Style=4"
190 DRAW.ARC.FILLED 128,160,80,80,45,135,RGB(255,0,0),5
200 DRAW.TEXT 0,128,160,"Style=5"
210 DRAW.ARC.FILLED 192,160,80,80,45,135,RGB(255,0,0),6
220 DRAW.TEXT 0,192,160,"Style=6"
230 DRAW.ARC.FILLED 256,160,80,80,45,135,RGB(255,0,0),7
240 DRAW.TEXT 0,256,160,"Style=7"
999 GOTO 999
```

And the resulting screen display:



DRAW.BOX x1, y1, x2, y2, color

Draws a box with corners at x1, y1 and x2, y2 using color.

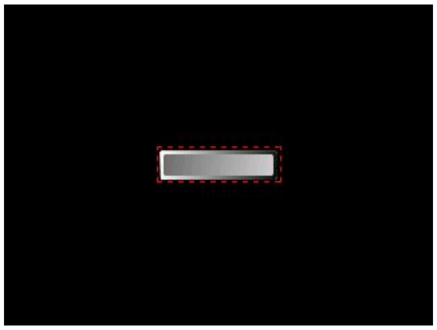
DRAW.BOX.DASHED x1, y1, x2, y2, color, pattern, scale

Draws a dashed line box with corners at x1, y1 and x2, y2 using color. Each bit in the pattern is used scale times to draw the pixels in the box lines; a pattern of 21845 (hex 5555) (binary 0101010101010101) with a scale of 4 would draw the box using 4 pixel dashes, 4 pixels apart.

Here's a sample that draws a 'crawling ants' flashing line around a bitmap button:

```
10 REM DRAW.BOX.DASHED demo
15 @ANSI.ENABLE = 0 : @BACKLIGHT = 1
20 xc = 160 : yc = 120 : w = BITMAP.WIDTH("ButtonK4.bmp") : h = BITMAP.HEIGHT("ButtonK4.bmp")
25 @SURFACE = 2 : DRAW.BITMAP xc - w/2, yc - h/2, "ButtonK4.bmp" : @SURFACE = 1
30 blx = xc-(w/2)-2 : bly = yc-(h/2)-2 : bux = xc+(w/2)+2 : buy = yc+(h/2)+2
35 ONEVENT @TIMER[0], GOSUB 1000
40 @TIMER[0] = 10
45 GOTO 45
1000 DRAW.COPY 2, blx, bly, blx, bly, w + 4, h + 4
1005 IF pattern = HEX.VAL("5555") THEN pattern = HEX.VAL("AAAA") ELSE pattern = HEX.VAL("5555")
1010 DRAW.BOX.DASHED blx, bly, bux, buy, RGB(255,0,0), pattern, 4
1020 DRAW.TOGGLE
1025 @TIMER[0] = 10
1030 RETURN
```

And the resulting screen display:



DRAW.BOX.FILLED x1, x2, y1, y2, color

Draws a filled box with corners at x1, y1 and x2, y2 using color.

DRAW.CIRCLE x, y, r, color

Draws a circle centered at x, y of radius r using color.

DRAW.ELLIPSE x, y, width, height, color

Draws an ellipse centered at x, y of width and height using color.

DRAW.ELLIPSE.FILLED x, y, width, height, color

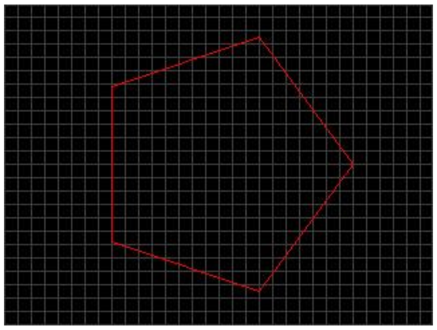
Draws a filled ellipse centered at x, y of width and height using color.

DRAW.POLYGON x[n], y[n],color

Draws an ‘n’ sided polygon with the x and y array of vertices using color where n ≥ 3. Here’s a sample program that draws regular polygons:

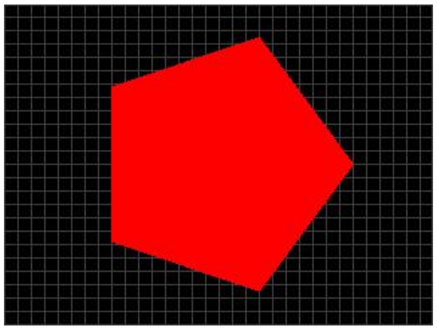
```
10 REM demo draw.polygon
15 @ANSI.ENABLE = 0 : @BACKLIGHT = 1
20 GOSUB 1000
25 INPUT "how many sides ? ", n : IF n <= 0 THEN STOP
30 GOSUB 1000
35 radius = 100 : xc = 160 : yc = 120
40 DIM x[n], y[n]
45 FOR i = 0 TO n-1
50 x[i] = MULDIV(radius, COS((360 * i) / n), 1024) + xc
55 y[i] = MULDIV(radius, SIN((360 * i) / n), 1024) + yc
60 NEXT i
65 DRAW.POLYGON x, y, RGB(255,0,0)
70 GOTO 25
1000 REM draw grid
1005 DRAW.FILL 0,0,319,239,RGB(0,0,0)
1010 FOR xg = 0 TO 320 STEP 10 : DRAW.LINE xg, 0, xg, 239, RGB(64,64,64) : NEXT xg
1015 DRAW.LINE 319,0,319,239,RGB(64,64,64)
1020 FOR yg = 0 TO 240 STEP 10 : DRAW.LINE 0, yg, 319, yg, RGB(64,64,64) : NEXT yg
1025 DRAW.LINE 0,239,319,239,RGB(64,64,64)
1030 RETURN
Ready
run
how many sides ? 5
how many sides ? 0
STOP in line 25
Ready
```

And the resulting screen display:



DRAW.POLYGON.FILLED x[n], y[n],color

Draws an ‘n’ sided filled polygon with the x and y array of vertices using color where n ≥ 3. Try changing line 65 in the DRAW.POLYGON example above:



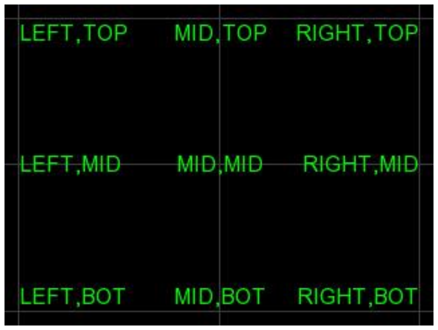
DRAW.TEXT font, x, y, expression

Draws the text string of the expression justified at x, y using the attributes of Font table entry font.

Here’s a sample program that shows how the font’s .HALIGN and .VALIGN system variables affect the text display:

```
10 REM demo DRAW.TEXT and @FONT.HALIGN, @FONT.VALIGN
15 @ANSI.ENABLE = 0 : @ANSI.CURSOR = 0 : @BACKLIGHT = 1
20 REM Draw grid
25 DRAW.LINE 10,0,10,239,RGB(64,64,64)
30 DRAW.LINE 160,0,160,239,RGB(64,64,64)
35 DRAW.LINE 309,0,309,239,RGB(64,64,64)
40 DRAW.LINE 0,10,319,10,RGB(64,64,64)
45 DRAW.LINE 0,120,319,120,RGB(64,64,64)
50 DRAW.LINE 0,229,319,229,RGB(64,64,64)
55 REM Draw aligned text in green
60 @FONT.FCOLOR[0] = RGB(0,255,0)
65 @FONT.HALIGN[0]=0:@FONT.VALIGN[0]=0:DRAW.TEXT 0,10,10,"LEFT,BOT"
70 @FONT.HALIGN[0]=1:@FONT.VALIGN[0]=0:DRAW.TEXT 0,160,10,"MID,BOT"
75 @FONT.HALIGN[0]=2:@FONT.VALIGN[0]=0:DRAW.TEXT 0,309,10,"RIGHT,BOT"
80 @FONT.HALIGN[0]=0:@FONT.VALIGN[0]=1:DRAW.TEXT 0,10,120,"LEFT,MID"
85 @FONT.HALIGN[0]=1:@FONT.VALIGN[0]=1:DRAW.TEXT 0,160,120,"MID,MID"
90 @FONT.HALIGN[0]=2:@FONT.VALIGN[0]=1:DRAW.TEXT 0,309,120,"RIGHT,MID"
95 @FONT.HALIGN[0]=0:@FONT.VALIGN[0]=2:DRAW.TEXT 0,10,229,"LEFT,TOP"
100 @FONT.HALIGN[0]=1:@FONT.VALIGN[0]=2:DRAW.TEXT 0,160,229,"MID,TOP"
105 @FONT.HALIGN[0]=2:@FONT.VALIGN[0]=2:DRAW.TEXT 0,309,229,"RIGHT,TOP"
110 GOTO 110
```

And the resulting screen display:



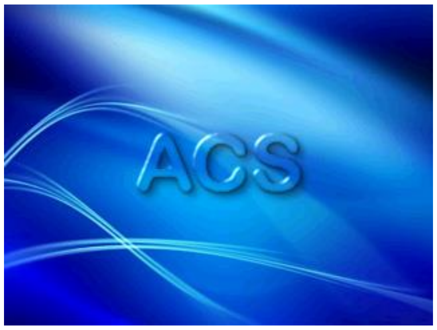
DRAW.BITMAP x, y, "imageResourceName"

Draws the named bitmap image from the resource table with the lower left corner located at x, y.

Here's a sample. The RESOURCES.LIST command is used to see what bitmap resources named starting with "Background". One of these named resources is then drawn on the display:

```
resources.list background*.bmp
Background_DateTimeScreen.bmp        153,676
Background_HomeScreenWithACSLogo.bmp 153,676
Background_PoolScreen.bmp            153,676
Background_TimerEditScreen.bmp       153,676
Background_TimersScreen.bmp          153,676
Ready
list
10 REM DRAW.BITMAP demo
15 @ANSI.ENABLE = 0 : @BACKLIGHT = 1
20 DRAW.BITMAP 0, 0, "Background_HomeScreenWithACSLogo.bmp"
25 GOTO 25
Ready
run
```

And the resulting screen:



DRAW.BITMAP.INDEXED index, x, y, "imageResourceName"

Draws a portion of the horizontally stacked bitmap image from the resource table selected by index at x, y. The stacked image must be square (based upon the image height). The number of instances or portions contained in the image equals the image width divided by the image height.

This command can be used to implement an animated image by changing the index and redrawing the image based upon on a timer event, or an indicator that can change its appearance upon some condition.

DRAW.BITMAP.TRANSP mode, color, x, y, "imageResourceName"

Draws a bitmap image from the resource table using transparency mode and color at x, y:

Where:

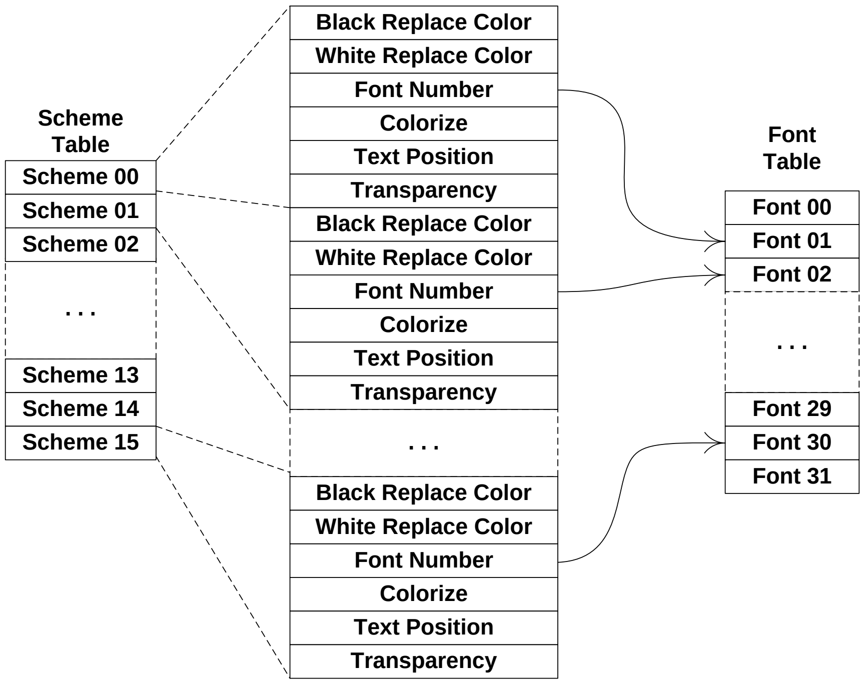
mode	Operation	color	Description
0	Magic Value	Magic index or magic color	For indexed .BMP images the index value corresponding to the magic color For 16bpp .BMP images the RGB565 magic color.
1	None	Not used	Surface pixel = image pixel
2	Darken	RGB565 color to be used as a multiplier	Surface pixel = surface pixel * RGB565 color (can darken image)
3	Source Color Filter	Not used	Surface pixel = surface pixel * image pixel (5:6:5 channel by channel multiply) (color filter from image)
4	Source Alpha Mask	RGB565 color to be used as a 'white' replacement	Surface pixel = (surface pixel * RGB565 color) + (1 – image pixel) * surface pixel (treats image as an alpha mask, white solid, black transparent, intermediate blended)

5	Fill	RGB565 fill color	Surface pixel = RGB565 color
6	Transparency Blend	RGB565 color to use as an Alpha multiplier	Surface pixel = (RGB565 color * image pixel) + (1 – RGB565 color) * surface pixel (blends image into surface by ratio determined by RGB565 color)
7	Shadow	RGB565 color to use as an Alpha multiplier	Surface pixel = (RGB565 color * image pixel * surface pixel) + (1 – image pixel) * surface pixel (image is mask to apply multiplication of RGB565 color)

SCHEMES

Schemes are implemented to provide a common way to label and colorize all higher level screen objects such as buttons, sliders and icons. Schemes are used in pairs with the first element of the pair used for rendering the screen object when it isn't touched, and the second used to render the object when it is touched. The two replacement colors can provide colorization of gray scale buttons that changes when the buttons are touched and released. The schemes are stored in a table and are referenced by the entry number.

There sixteen entries in the Schemes table yielding eight scheme pairs.



The following commands are provided to manage the contents of the Schemes table:

SCHEMES.INIT

Initializes the Schemes table.

SCHEMES.LIST {#N} {start{, end}} … SCHEMES.LIST {start{-end}}

Lists the contents of the Schemes table, optionally to a previously opened file {#N}.

SCHEMES.LOAD filename{.schemes}

Loads the Schemes table from a JSON formatted file on the micro SD card. (***See the appendix JSON File Formats in the Color 320x240 LCD Display Terminal User's Manual)***

SCHEMES.SAVE filename{.schemes}

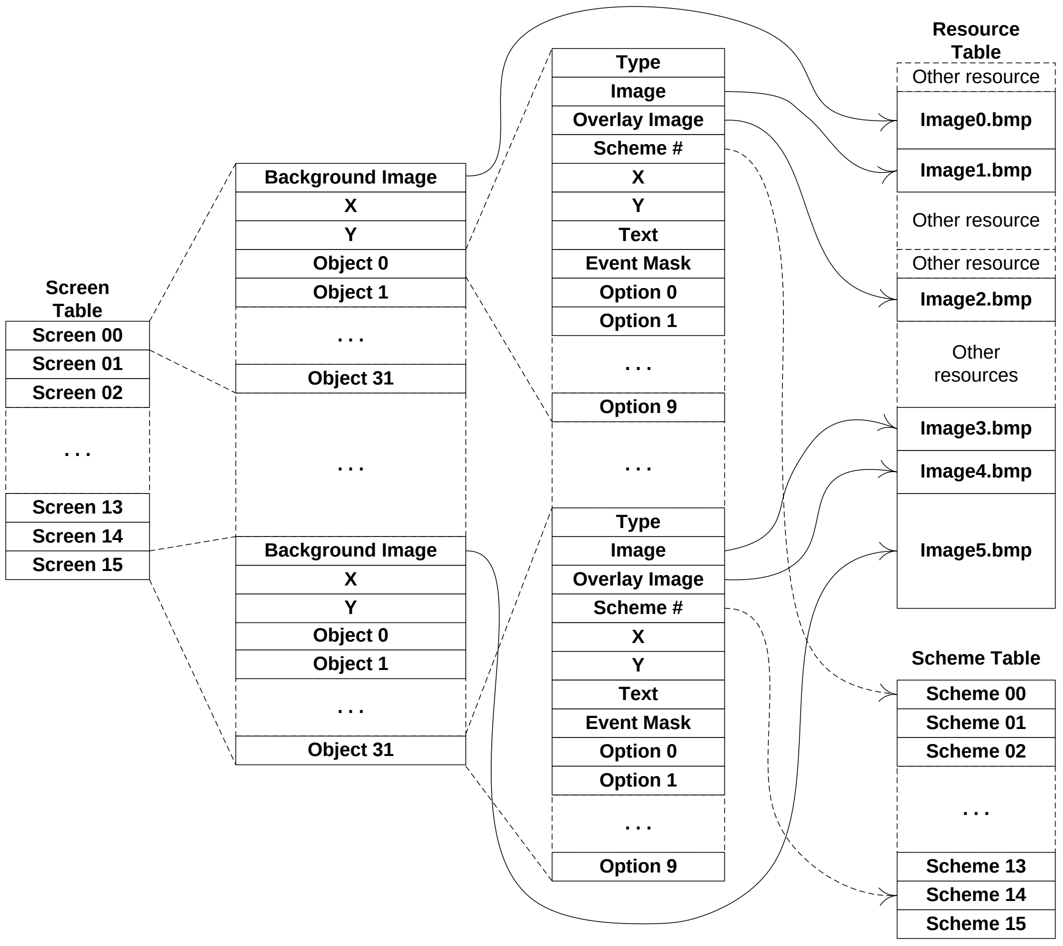
Saves the Schemes table to a JSON formatted file on the micro SD card. ***(See the appendix JSON File Formats in the Color 320x240 LCD Display Terminal User's Manual)***

SCREENS

Screens are implemented to provide a grouped collection of placed screen objects such as buttons, sliders and icons overlaid on top of a background image. Screens are drawn entirely by the display including the automatic colorization of objects as they are manipulated with the interactions reported as events.

The screens may be navigated to directly by their number or like a stack – last in, first out. Events are fired when screens are navigated to or away from. The screens are stored in a table and are referred to by their entry number.

There are sixteen entries in the Screen table and each screen can contain up to thirty two screen objects. Object entries in the Screen table must be contiguous – screen objects will not be processed beyond the first zero entry.



The individual fields in the Screens table are accessed via the @SCREEN system variables which are indexed by the screen number.

The individual fields in the Screen Objects in each screen are accessed via the @SCREEN.OBJ system variables which are indexed by both the screen number and screen object number.

The following commands are provided to manage the contents of the Screens table:

SCREENS.INIT

Initializes the Screens table.

SCREENS.LIST

Lists the contents of the Screens table.

SCREENS.LOAD filename{.screens}

Loads the Screens table from a JSON formatted file on the micro SD card. (***See the appendix JSON File Formats in the Color 320x240 LCD Display Terminal User's Manual)***

SCREENS.SAVE filename{.screens}

Saves the Screens table to a JSON formatted file on the micro SD card. (***See the appendix JSON File Formats in the Color 320x240 LCD Display Terminal User's Manual)***

The following commands are provided to operate the Screen system:

SCREENS.CHANGETO screenNumber

Constructs and displays the contents of the Screens table screenNumber entry. Changing to the first screen is used to startup the operation of the Screens system. Before any DRAWing command can be used after a SCREENS.CHANGETO the screen has to be fully constructed as indicated by the @SCREEN.EVENT system variable / event being signaled.

SCREENS.PUSHTO screenNumber

Saves the currently display Screens table entry number on a stack, deconstructs the current screen and then constructs and displays the contents of the of the Screens table screenNumber entry.

SCREENS.POP

Deconstructs the current screen and then restores the last pushed to Screens table entry number from the stack then constructs and displays it.

Events

ACS Basic provides the concept of an ***Event***. Events occur outside of the normal program execution flow and are processed in between the execution of individual program statements. Some system variables have ***Events*** associated with them and may be referenced in **ONEVENT, SIGNAL** and **WAIT** statements. There are three ways to process an event: asynchronously with an **ONEVENT** handler, synchronously with a **WAIT** statement or by polling the system variable's value in the program to see when the event occurs.

In order to process an event asynchronously, Basic has to be informed of what code to execute when a certain event occurs. This is done using the **ONEVENT** statement. After Basic executes each program statement, it scans the table of events looking to see if any have been signaled. If an **ONEVENT** handler for a signaled event has been specified by the program, then Basic will force a subroutine call to the event handler before the next program statement is executed.

Events have an implicit priority with higher priority events being able to interrupt execution of lower priority event handlers. Here's an example of touch event handling:

```
100 REM Test Touch Events
110 ONEVENT @TOUCH.EVENT, GOSUB 1000
130 GOTO 130
1000 T=@TOUCH.EVENT: X=@TOUCH.X: Y=@TOUCH.Y
1010 ON T, GOTO 1015,1020,1025,1030
1015 RETURN
1020 PRINT "Touch   @ ";X;", ";Y : RETURN
1025 PRINT "Move    @ ";X;", ";Y : RETURN
1030 PRINT "Release @ ";X;", ";Y : RETURN
1035 RETURN
Ready
```

This would print **"Touch @ x, y", "Move @ x, y", or "Release @ x, y"** on the screen as the user interacts with the touchscreen.

In order to handle an event synchronously a program may wait for an event to occur by using the **WAIT** statement. Program execution stalls at that statement until the specified event happens. Alternatively, the program may poll the associated system variable's value in a loop looking for the event to have been signaled. Here's an example of polling for the @SECOND system variable to change:

```
10 REM poll @SECOND
15 Seconds = @SECOND
20 LIF Seconds <> LastSeconds THEN PRINT Seconds : LastSeconds = Seconds
25 GOTO 15
Ready
run
54
55
56
57
 ESC at line 15
Ready
```

This would print the value of @SECOND every time it changes.

The **SIGNAL** statement may be used in a program to force an event to happen.

It is very important to note that the **ONEVENT** handler subroutine executes in the context of the running program: it has access to all program variables. Since the event handler may be executed at any time in between any program statements care should be used when changing program variables from within an event handler as it may cause unexpected results in the execution of other program statements that may be using and depending upon the values of those same variables. Incorrect or unexpected program execution may result – code event handlers carefully.

See the **ONEVENT** statement definition below for a table showing what events may be processed and listing their relative priority.

User Defined Functions

ACS Basic also provides the ability for the user to write and call functions that are defined using statements in the program. Two commands provide this capability; **FUNCTION** and **ENDFUNCTION**.

The **FUNCTION** command starts a function definition. It is followed by a variable name that also identifies its type as integer or string ($). The function variable name is then followed by a parenthesized list of zero or more parameter variables. The name of the function becomes a defined variable global to the entire program. The parameter variables will be created and assigned values when the function is subsequently called. Any additional statements following on the same line are ignored. Here are a couple of examples:

`100 FUNCTION MyFunction(parm1, parm2$)`	Defines an integer function named MyFunction that takes two arguments; an integer named parm1 and a string named parm2$
`100 FUNCTION Test$()`	Defines a string function named Test$ that takes no arguments

The **ENDFUNCTION** command ends a function definition. Any additional statements following on the same line are ignored. When the defined function is called the **ENDFUNCTION** command behaves like a **RETURN** command – execution continues after from where the **FUNCTION** was called.

FUNCTIONs may be located at the beginning of the program, or elsewhere. After they have been defined the functions may be called to execute the commands that they encompass.

The function call actually invokes a defined function by creating the function's parameter list variables, assigning the matching argument list expressions to them, then executing the function's statements as a subroutine. The function argument variables and any additional variables that are created inside the function are valid throughout the execution of the function (and any nested function calls) and are discarded upon execution of the **ENDFUNCTION** command.

```
10 MyFunction(10, “test”)
```

Values may be passed to a function in three ways; through the function call argument list, through the global function's variable name or through any other variable defined outside of the execution of the function.

Values may be returned from a function in two ways; through the global function's variable name or through any other variable defined outside of the execution of the function.

The operation of user functions is better shown with some examples. In this first example a simple integer function named **test( )** is defined that takes three arguments and adds them together with the sum returned through the function name by assignment in line 110. Lines 100 through 115 define the function. In lines 15 and 20 the function **test( )** is called twice with different arguments:

```
10 REM integer function
15 test(1,2,3):PRINT "= ";test
20 test(4,5,6):PRINT "= ";test
25 END
100 FUNCTION test(arg1,arg2,arg3)
105  PRINT "test(";arg1;",";arg2;",";arg3;") ";
110  test=arg1+arg2+arg3
115 ENDFUNCTION
Ready
run
test(1,2,3) = 6
test(4,5,6) = 15
Ready
vars
test -> r/w IntFunction  = 15 @ line 100
Ready
```

The first thing to observe is if we **STOP** the program inside of the **FUNCTION** you can see the parameter variables that receive values from the function arguments. These parameter variables are discarded when the function returns from the function call at the **ENDFUNCTION** statement:

```
112 stop
list
10 REM integer function
15 test(1,2,3):PRINT "= ";test
20 test(4,5,6):PRINT "= ";test
25 END
100 FUNCTION test(arg1, arg2, arg3)
105  PRINT "test(";arg1;","; arg2;","; arg3;") ";
110  test=arg1+arg2+arg3
112 STOP
115 ENDFUNCTION
Ready
run
test(1,2,3) STOP in line 112
Ready
vars
test   -> r/w IntFunction  = 6 @ line 100
 arg1  -> r/w Int          = 1
 arg2  -> r/w Int          = 2
 arg3  -> r/w Int          = 3
Ready
```

In the prior example, in lines 15 and 20 we are calling the test( ) function and then PRINTing its value. This can be combined into a single step by calling the function with arguments from the PRINT statement:

```
list
10 REM integer function
15 PRINT "= ";test(1,2,3)
20 PRINT "= ";test(4,5,6)
25 END
100 FUNCTION test(arg1,arg2,arg3)
105  PRINT "test(";arg1;",";arg2;",";arg3;") ";
110  test=arg1+arg2+arg3
115 ENDFUNCTION
Ready
run
= test(1,2,3) 6
= test(4,5,6) 15
Ready
```

Here's an example of a similar string function named test$( ) that takes three string arguments and concatenates them together with the result returned by assignment to the function name:

```
10 REM string function
15 PRINT "= ";test$("one","two","three")
20 PRINT "= ";test$("four","five","six")
25 END
100 FUNCTION test$(arg1$,arg2$,arg3$)
105  PRINT "test$(";arg1$;",";arg2$;",";arg3$;") ";
110  test$=arg1$+arg2$+arg3$
115 ENDFUNCTION
Ready
run
= test$(one,two,three) onetwothree
= test$(four,five,six) fourfivesix
Ready
```

The following example is a string function named test$( ) that takes a mixed string / integer parameter list – the first parameter named in$ passes in the string value and the second parameter named howmany tells the function how many times to concatenate the input string with itself. Notice how the function variable is global and retains its value outside of the function – in this case causing an unintended side effect of the prior call's result being prefixed to the second function call's result:

```
10 REM functions
15 PRINT "= ";test$("A",10)
20 PRINT "= ";test$("abc",20)
```

```
25 END
100 FUNCTION test$(in$, howmany)
105  PRINT "test$(";in$;",";howmany;") ";
110  FOR count=1 TO howmany:test$=test$+in$:NEXT count
115 ENDFUNCTION
Ready
run
= test$(A,10) AAAAAAAAAA
= test$(abc,20) AAAAAAAAAAabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc
Ready
```

Function calls may also be nested – you can call a function from within a function.

In the following example the function cube$( ) calls function cube( ) to obtain the integer value of its argument cubed, then converts that to a string that is PRINTed. Note that the FUNCTION definitions can be in any order as they are identified and defined when the program starts:

```
10 REM functions
40 PRINT cube$(10)
50 END
100 REM define functions
105 FUNCTION cube(value)
110  cube=value*value*value
115 ENDFUNCTION
120 FUNCTION cube$(value)
125  cube$=STR$(cube(value))
130 ENDFUNCTION
Ready
run
1000
Ready
vars
cube  -> r/w IntFunction  = 1000 @ line 105
cube$ -> r/w Str$Function = "1000" @ line 120
Ready
```

Functions can be dynamically redefined – as long as the type (integer / string) remains the same. In this example the compute( ) function is redefined to calculate the square of its argument the first time, then the cube of its argument the next time by executing through it using a GOSUB/RETURN:

```
10 REM function redefinition
20 GOSUB 1000 : REM define compute FUNCTION as square
30 PRINT compute(10)
40 GOSUB 1100 : REM define compute FUNCTION as cube
50 PRINT compute(10)
60 END
1000 REM compute square
1010 FUNCTION compute(value)
1020  compute = value * value
1030 ENDFUNCTION
1040 RETURN : REM required for GOSUB redefinition
1100 REM compute cube
1110 FUNCTION compute(value)
1120  compute = value * value * value
1130 ENDFUNCTION
1140 RETURN : REM required for GOSUB redefinition
Ready
run
100
1000
Ready
vars
compute -> r/w IntFunction  = 1000 @ line 1110
Ready
```

Another example of functions calling functions:

```
10 REM function nesting
30 PRINT ten(2)
999 END
1000 FUNCTION ten(value)
1010  nine(value) : ten = value * nine : PRINT "nine ",
1020 ENDFUNCTION
1030 FUNCTION nine(value)
1040  eight(value) : nine = value * eight : PRINT "eight ",
1050 ENDFUNCTION
1060 FUNCTION eight(value)
1070  seven(value) : eight = value * seven : PRINT "seven ",
1080 ENDFUNCTION
1090 FUNCTION seven(value)
1100  six(value) : seven = value * six : PRINT "six ",
1110 ENDFUNCTION
1120 FUNCTION six(value)
1130  five(value) : six = value * five : PRINT "five ",
1140 ENDFUNCTION
1150 FUNCTION five(value)
1160  four(value) : five = value * four : PRINT "four ",
1170 ENDFUNCTION
1180 FUNCTION four(value)
1190  three(value) : four = value * three : PRINT "three ",
1200 ENDFUNCTION
1210 FUNCTION three(value)
1220  two(value) : three = value * two : PRINT "two ",
1230 ENDFUNCTION
1240 FUNCTION two(value)
1250  one(value) : two = value * one : PRINT "one ",
1260 ENDFUNCTION
1270 FUNCTION one(value)
1280  one = value
1290 ENDFUNCTION
Ready
run
one two three four five six seven eight nine 1024
Ready
vars
ten   -> r/w IntFunction  = 1024 @ line 1000
nine  -> r/w IntFunction  = 512 @ line 1030
eight -> r/w IntFunction  = 256 @ line 1060
seven -> r/w IntFunction  = 128 @ line 1090
six   -> r/w IntFunction  = 64 @ line 1120
five  -> r/w IntFunction  = 32 @ line 1150
four  -> r/w IntFunction  = 16 @ line 1180
three -> r/w IntFunction  = 8 @ line 1210
two   -> r/w IntFunction  = 4 @ line 1240
one   -> r/w IntFunction  = 2 @ line 1270
Ready
```

And again – if we STOP the program in the deepest nested function you can see the nested function arguments – notice that each function uses a parameter named value but these value argument variables are local to each function. This is shown in the following vars command output by the indentation:

```
1285 stop
list
10 REM function nesting
30 PRINT ten(2)
999 END
1000 FUNCTION ten(value)
1010  nine(value) : ten = value * nine : PRINT "nine ",
1020 ENDFUNCTION
1030 FUNCTION nine(value)
1040  eight(value) : nine = value * eight : PRINT "eight ",
1050 ENDFUNCTION
1060 FUNCTION eight(value)
1070  seven(value) : eight = value * seven : PRINT "seven ",
1080 ENDFUNCTION
1090 FUNCTION seven(value)
1100  six(value) : seven = value * six : PRINT "six ",
1110 ENDFUNCTION
1120 FUNCTION six(value)
1130  five(value) : six = value * five : PRINT "five ",
1140 ENDFUNCTION
1150 FUNCTION five(value)
1160  four(value) : five = value * four : PRINT "four ",
1170 ENDFUNCTION
```

```
1180 FUNCTION four(value)
1190  three(value) : four = value * three : PRINT "three ",
1200 ENDFUNCTION
1210 FUNCTION three(value)
1220  two(value) : three = value * two : PRINT "two ",
1230 ENDFUNCTION
1240 FUNCTION two(value)
1250  one(value) : two = value * one : PRINT "one ",
1260 ENDFUNCTION
1270 FUNCTION one(value)
1280  one = value
1285 STOP
1290 ENDFUNCTION
Ready
run
STOP in line 1285
Ready
vars
ten             -> r/w IntFunction  = 0 @ line 1000
nine            -> r/w IntFunction  = 0 @ line 1030
eight           -> r/w IntFunction  = 0 @ line 1060
seven           -> r/w IntFunction  = 0 @ line 1090
six             -> r/w IntFunction  = 0 @ line 1120
five            -> r/w IntFunction  = 0 @ line 1150
four            -> r/w IntFunction  = 0 @ line 1180
three           -> r/w IntFunction  = 0 @ line 1210
two             -> r/w IntFunction  = 0 @ line 1240
one             -> r/w IntFunction  = 2 @ line 1270
 value          -> r/w Int          = 2
  value         -> r/w Int          = 2
   value        -> r/w Int          = 2
    value       -> r/w Int          = 2
     value      -> r/w Int          = 2
      value     -> r/w Int          = 2
       value    -> r/w Int          = 2
        value   -> r/w Int          = 2
         value  -> r/w Int          = 2
          value -> r/w Int          = 2
Ready
```

Errors

The following errors can be produced. The placeholder ‘dd’ in the message is replaced with the line number where the error was detected if the error was encountered in a running program. Some Syntax Errors will provide additional information after the line number further identifying the error:

Error #	Error Message	Causes
1	"Syntax error in line dd"	Incorrect statement format
2	"Illegal program command error in line dd"	Direct mode only statement in program mode
3	"Illegal direct command error in line dd"	Program mode only statement in direct mode
4	"Line number error in line dd"	Target line number not in program
5	"Wrong expression type error in line dd"	Numeric value when String expected or vice versa
6	"Divide by zero error in line dd"	Division by zero
7	"Nesting error in line dd "	NEXT without preceding FOR, RETURN without preceding GOSUB
8	"File not open error in line dd "	CLOSE#, LIST#, PRINT# or INPUT# without successful OPEN statement
9	"File already open error in line dd "	OPEN# on already open file
10	“File # Out of Range in line dd”	File # out of range 0 - 23
11	"Input error in line dd "	Numeric value expected in INPUT # statement
12	"Dimension error in line dd "	Dimension error
13	“Index out of range in line dd”	Subscript out of range
14	"Data error in line dd "	ORDER line # not DATA statement, READ past DATA statements
15	"Out of memory error in line dd "	Insufficient memory
16	"No File System error in line dd "	ACS Basic running without CF card
17	“Unknown @var error in line dd “	Unknown system variable
18	"Timer # out of range error in line dd "	@TIMER(x) subscript out of range 0 - 9
19	"Port # out of range error in line dd "	@PORT(x) subscript out of range 0 - 255
20	"Contact # out of range error in line dd "	@CONTACT(x), @CLOSURE(x), @OPENING(x) subscript out of range
21	"Stack Overflow error in line dd "	Too many nested FOR and/or GOSUB and/or events
22	"No SD card error in line dd "	Statement requiring microSD card with no card detected
23	"Invalid .WAV file error in line dd "	.WAV file format not 44.1KHz 16-bit mono or stereo or @SOUND$ queue full
24	“DRAW.x arguments Out of Range error in line dd”	One or more argument to a DRAW.x statement are out of range
25	“FWRITE record # Out of Range error in line dd”	Attempt to FWRITE to a record number that is past the immediate end of file
26	“FWRITE exceeds record length error in line dd”	Length of data in FWRITE variables list including commas and quotes exceeds the recordlength specified in the associated FOPEN
27	“FINSERT record # Out of Range error in line dd”	Attempt to FINSERT to a record number that is past the immediate end of file
28	“FINSERT exceeds record length error in line dd”	Length of data in FINSERT variables list including commas and quotes exceeds the recordlength specified in the associated FOPEN
29	“FDELETE past end of file error in line dd ”	FDELETE record number exceeds file length
30	“Can’t delete file error in line dd”	Can’t delete file
31	“Can’t make directory error in line dd”	Can’t create directory
32	“Can’t rename file error in line dd”	Can’t rename file
33	“Unknown Command error in line dd”	ACS Basic doesn’t recognize the command
34	“Can’t use @VAR in line dd”	Illegal use of systemvar in FOR, DIM, INPUT, READ, FREAD or FINPUT statement
35	“Mis-matched quotes in line dd”	Missing one of a pair of double quotes delimiting a string
36	“RGB” argument error”	Problem with an argument to the RGB( ) function
37	“Unsupported bitmap file”	Problem with filename argument to the DRAW.BITMAP statement
38	“FREAD record # out of range	
39	“Resource not found”	
40	“Resource already exists”	
41	“Font # out of range”	
42	“.fonts file invalid”	
43	“Scheme # out of range”	
44	“.schemes file invalid”	
45	“Obj # out of range”	
46	“Screen # out of range”	
47	“.screens file invalid”	
48	“Config # out of range”	
49	“Config Item < min or > max”	
50	“DRAW.POLYGON”	

51	“uSD Card”	
52	“File System”	
53	“Read Only”	Attempt to write to a CONST variable
54	“Option # Out of Range”	
55	“Data # Out of Range”	
56	ACS Internal Usage	
57 - 32767	“x error in line dd”	ERROR x statement

Extended Characters

The following characters may be displayed by sending the multiple decimal byte sequence with CHR$( ):

Ç 195, 135	ü 195, 188	é 195, 169	â 195, 162	ä 195, 164	à 195, 160	å 195, 165	ç 195, 167
ê 195, 170	ë 195, 171	è 195, 168	ï 195, 175	î 195, 174	ì 195, 172	Ä 195, 132	Å 195, 133
É 195, 137	æ 195, 166	Æ 195, 134	ô 195, 180	ö 195, 182	ò 195, 178	û 195, 187	ù 195, 185
ÿ 195, 191	Ö 195, 150	Ü 195, 156	¢ 194, 162	£ 194, 163	¥ 194, 165	₧ 226, 130, 167	ƒ 198, 146
á 195, 161	í 195, 173	ó 195, 179	ú 195, 186	ñ 195, 177	Ñ 195, 145	ª 194, 170	º 194, 186
¿ 194, 191	⌐ 226, 140, 144	¬ 194, 172	½ 194, 189	¼ 194, 188	¡ 194, 161	« 194, 171	» 194, 187
░ 226, 150, 145	▒ 226, 150, 146	▓ 226, 150, 147	│ 226, 148, 130	┤ 226, 148, 164	╡ 226, 149, 161	╢ 226, 149, 162	╖ 226, 149, 150
╕ 226, 149, 149	╣ 226, 149, 163	║ 226, 149, 145	╗ 226, 149, 151	╝ 226, 149, 157	╜ 226, 149, 156	╛ 226, 149, 155	┐ 226, 148, 144
└ 226, 148, 148	┴ 226, 148, 180	┬ 226, 148, 156	├ 226, 148, 156	─ 226, 148, 128	┼ 226, 148, 188	╞ 226, 149, 158	╟ 226, 149, 159
╚ 226, 149, 154	╔ 226, 149, 148	╩ 226, 149, 169	╦ 226, 149, 166	╠ 226, 149, 160	═ 226, 149, 144	╬ 226, 149, 172	╧ 226, 149, 167
╤ 226, 149, 164	╥ 226, 149, 165	╙ 226, 149, 153	╘ 226, 149, 152	╒ 226, 149, 146	╓ 226, 149, 147	╫ 226, 149, 171	╪ 226, 149, 170
┘ 226, 148, 152	┌ 226, 148, 140	█ 226, 150, 136	▄ 226, 150, 132	▌ 226, 150, 140	▐ 226, 150, 144	▀ 226, 150, 128	α 195, 177
ß 195, 159	Γ 206, 147	π 207, 128	Σ 206, 163	σ 207, 131	µ 194, 181	τ 207, 132	Φ 206, 166
Θ 206, 152	Ω 206, 169	δ 206, 180	∞ 226, 136, 158	φ 207, 134	ε 206, 181	∩ 226, 136, 169	≡ 226, 137, 161
± 194, 177	≥ 226, 137, 165	≤ 226, 137, 164	⌠ 226, 140, 160	⌡ 226, 140, 161	÷ 195, 183	≈ 226, 137, 136	° 194, 176

· 194, 183	√ 226, 136, 154	ⁿ 226, 129, 191	² 194, 178	■ 226, 150, 160	← 226, 134, 144	↑ 226, 134, 145	→ 226, 134, 146
↓ 226, 134, 147							

Socket Programming

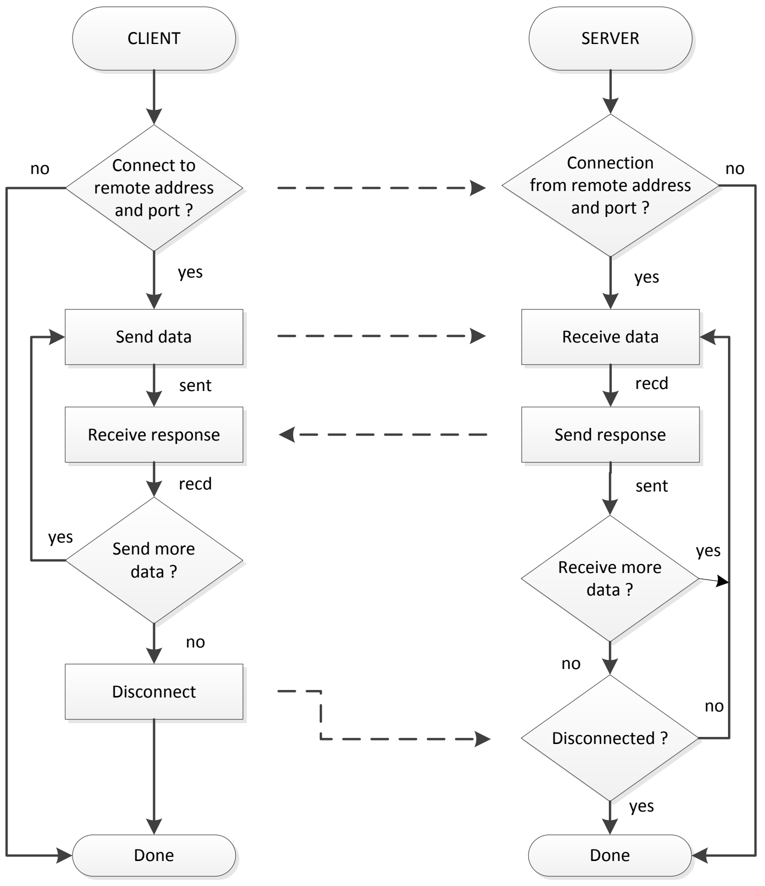
ACS Basic provides a set of primitives that allow networked devices to exchange messages using a pre-arranged protocol. One or more devices can communicate using TCP/IP over the Ethernet interface in a client/server fashion. The implementation of this network communication is referred to as "socket programming".

This network socket programming achieves communication between two machine applications using a client/server methodology – one device's application establishes itself as a network server that 'listens' for incoming TCP/IP connections on a specified port number. The other device's application implements a network client that 'connects' to a listening server with subsequent message exchange. The client device then 'disconnects' from the server and the process can repeat.

ACS Basic Sockets

The ACS Basic socket implementation is highly simplified and half-duplex – communication ping-pongs back and forth between the client and server devices. The Client device initiates a connection to a listening Server device and after the connection is established sends the first message then receives a response. The Server device listens for a connecting Client device and after connection is established receives the first message then sends a response. This is slightly different from conventional socket programming where each side can send and receive at the same time.

The following diagram is a simplified representation of the communication sequence:



There are two styles of socket programming – blocking or synchronous and non-blocking or asynchronous. ACS Basic has statements and functions to facilitate both styles. Synchronous sockets do not allow any other program statements to be executed while the socket is connected, sending or receiving – program execution is totally consumed by the socket command. Asynchronous sockets initiate the socket operation and then return program execution to the program – the status of the socket command is returned via a system event variable.

Blocking Sockets

Synchronous or blocking socket operations are implemented using both built-in and user defined functions. The built-in function's return value provides the program with the status of the operation. The user defined functions are called by the built-in function to provide control points during the socket operation. There are two blocking functions to implement the client or server on a device.

Client Blocking Connection

The synchronous socket client connection takes the form:

```
SOCKET.SYNC.CONNECT(#N, "ip:port", connect( ), send( ), recv( ))
```

#N	File number
"ip:port"	String representation of IPv4 address and TCP/IP port number
connect( )	User function called when connection occurs
send( )	User function called to send data to 'file' #N
recv( )	User function called to receive data from 'file' #N

The **SOCKET.SYNC.CONNECT( )** function opens the TCP/IP network connection as an open file **#N** to the remote device using the **"ip.port"** string.

- When the connection is established the user provided **connect( )** function is called.
- When this function returns, the user provided **send( )** function is called to output data to the remote device using **PRINT #N** or **FPRINT #N** statements. The **send( )** function should return one to wait for received data, two to send more data or zero to disconnect.
- Next the user provided **recv( )** function is called to input data from the remote device using **INPUT #N** or **FINPUT #N** statements. The **recv( )** function should return one to send more data, two to receive more data or zero to disconnect.

The process repeats between **send( )** and **recv( )** calls until one of the functions returns zero or an error condition occurs. The **SOCKET.SYNC.CONNECT( )** function then returns a numeric value representing the "result" of the connection:

SOCKET.SYNC.CONNECT( ) returns	Description
0	Unknown / No Status
1	Disconnect / Done / No Error
2	Open Error – requested file **#N** failed to open
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT[#N]** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT[#N]** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT[#N] interval**

Server Blocking Listen

The synchronous socket server listen takes the form:

```
SOCKET.SYNC.LISTEN(#N, “:port”, connect( ), recv( ), send( ))
```

#N	File number
“:port”	String representation of IPv4 address and TCP/IP port number
connect( )	User function called when connection occurs
recv( )	User function called to receive data from ‘file’ #N
send( )	User function called to send data to ‘file’ #N

The **SOCKET.SYNC.LISTEN( )** function opens a TCP/IP port as file **#N** using the **“:port”** string.

- When a remote client establishes a connection to the listening port the user provided **connect( )** function is called.
- When this function returns, the user provided **recv( )** function is called to input data from the remote device using **INPUT #N** or **FINPUT #N** statements. The **recv( )** function should return one to send more data, two to receive more data or zero to disconnect.
- Next the user provided **send( )** function is called to output data to the remote device using **PRINT #N** or **FPRINT #N** statements. The **send( )** function should return one to wait for received data, two to send more data or zero to disconnect.

The process repeats between **recv( )** and **send( )** calls until one of the functions returns zero or an error condition occurs. The **SOCKET.SYNC.LISTEN( )** function then returns a numeric value representing the “result” of the connection:

SOCKET.SYNC.LISTEN( ) returns	Description
0	Unknown / No Status
1	Disconnect / Done / No Error
2	Open Error – requested file **#N** failed to open
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT[#N]** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT[#N]** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT[#N] interval**

Non-blocking Sockets

Asynchronous or non-blocking sockets are implemented using a statement to ‘start’ the communication, user functions to control the order and flow of data and a system variable to return the “result” of the connection. The user defined functions are called in-between other program statements that are executing to provide control points during the socket operation. There are two statements to ‘start’ the asynchronous socket communication.

Client Non-blocking Connection

The asynchronous client connection is started using the following program statement:

```
SOCKET.ASYNC.CONNECT #N, “ip:port”, connect( ), send( ), recv( )
```

#N	File number
“ip:port”	String representation of IPv4 address and TCP/IP port number
connect( )	User function called when connection occurs
send( )	User function called to send data to ‘file’ #N

recv( )	User function called to receive data from 'file' #N

The **SOCKET.ASYNC.CONNECT** statement opens the TCP/IP network connection as an open file **#N** to the remote device using the **"ip.port"** string. If this is successful execution continues with the following program statements.

- When the connection is established the user provided **connect( )** function is called.
- When this function returns, the user provided **send( )** function is called to output data to the remote device using **PRINT #N** or **FPRINT #N** statements. The **send( )** function should return one to wait for received data, two to send more data or zero to disconnect.
- Next the user provided **recv( )** function is called to input data from the remote device using **INPUT #N** or **FINPUT #N** statements. The **recv( )** function should return one to send more data, two to receive more data or zero to disconnect.

The process repeats between **send( )** and **recv( )** calls until one of the functions returns zero or an error condition occurs. The @**SOCKET.EVENT[#N}** system variable then fires and event and returns a numeric value representing the "result" of the connection:

@SOCKET.EVENT[#N]	Description
0	Unknown / No Status
1	Disconnect / Done / No Error
2	n/a – the 'open' was done initially
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT[#N]** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT[#N]** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT[#N] interval**

Server Non-blocking Listen

The asynchronous socket server listen takes the form:

```
SOCKET.ASYNC.LISTEN #N, ":port", connect( ), recv( ), send( )
```

#N	File number
":port"	String representation of IPv4 address and TCP/IP port number
connect( )	User function called when connection occurs
recv( )	User function called to receive data from 'file' #N
send( )	User function called to send data to 'file' #N

The **SOCKET.ASYNC.LISTEN** statement opens a TCP/IP port as an open file **#N** using the **":port"** string. . If this is successful execution continues with the following program statements.

- When a remote client establishes a connection to the listening port the user provided **connect( )** function is called.
- When this function returns, the user provided **recv( )** function is called to input data from the remote device using **INPUT #N** or **FINPUT #N** statements. The **recv( )** function should return one to send more data, two to receive more data or zero to disconnect.
- Next the user provided **send( )** function is called to output data to the remote device using **PRINT #N** or **FPRINT #N** statements. The **send( )** function should return one to wait for received data, two to send more data or zero to disconnect.

The process repeats between **recv( )** and **send( )** calls until one of the functions returns zero or an error condition occurs. The **SOCKET.SYNC.LISTEN( )** function then returns a numeric value representing the "result" of the connection:

@SOCKET.EVENT[#N]	Description
0	Unknown / No Status
1	Disconnect / Done / No Error
2	n/a – the file 'open' was done initially
3	Connection Timeout – no connection within the **@SOCKET.TIMEOUT[#N]** interval
4	Data Send Timeout – no send data acknowledgment within the **@SOCKET.TIMEOUT[#N]** interval
5	Receive Data Timeout – no received data within the **@SOCKET.TIMEOUT[#N] interval**

Communication Protocol

In order for meaningful communication to take place both the Client and Server devices have to agree on the exchanged message sequence and format. In the ACS Basic implementation messages are sent using the **PRINT #N** or **FPRINT #N** statements and messages are received using the **INPUT #N** or **FINPUT #N** statements. A message is considered to be "received" when the trailing carriage return from the sending **PRINT** statement is seen by the corresponding **INPUT** statement.

Messages can be constructed of multiple variables or constants, but there must be an agreement between the client and server. The **PRINT #N / INPUT #N** combination allows the value of a single variable to be communicated. The type of the variable must match between the **PRINT #N** on one device and the **INPUT #N** on the other device.

The **FPRINT #N / FINPUT #N** combination allows the values of multiple variables to be communicated. The order and type of the variables must match between the **FPRINT #N** on one device and the **FINPUT #N** on the other device. While an output message can be constructed using multiple **FPRINT #N** statements with trailing semi-colons except for the last one the received input message must be handled by a single **FINPUT #N** statement. The maximum message size is limited to the maximum size of a string variable.

The sequence of messages must also be defined although the simplified half-duplex implementation mandates that the connecting client device sends data first then receives and the listening server device receives data first then sends. Multiple messages can be sent in each direction at a time as long as there is an agreed upon message value such as an empty message that can be used to switch direction.

Either end can disconnect first but if the client sends data that isn't acknowledged or is waiting to receive data, a timeout will occur. If the server sends data that isn't acknowledged a timeout will occur; however if the server is waiting to receive data and the client disconnects before the timeout this is not considered to be an error as it is the result of a normal client disconnection.

Socket Examples

Here are some simple examples showing client and server, blocking and non-blocking. Note that because the underlying communications protocol of using **FINPUT #N var, var2$** and **FPRINT #N var, var2$** are the same that the sample blocking client can call the sample blocking or non-blocking server and vice versa.

Blocking Client

The following program is an example of a simple synchronous or blocking client. It is connecting to a server program on another device that essentially echoes back what it has received until the connection is terminated.

Line 1005 uses an **ON GOTO** statement to print the result of the **SOCKET.SYNC.CONNECT( )** function.

The **send( )** user defined function in lines 1065-1075 outputs an incrementing number n and a random numeric string as long as n < 10 and returns the non-zero value one. When n >= 10 the **send( )** function returns the zero value which causes the **SOCKET.SYNC.CONNECT( )** function to disconnect and exit.

The **recv( )** user defined function receives the echoed values and compares the returned string with what was sent. If they are identical the **recv( )** function returns the non-zero value one and the **SOCKET.SYNC.CONNECT( )** function continues. If there isn't a match the **recv( )** function returns the zero value and the **SOCKET.SYNC.CONNECT( )** function exits.

```
10 REM test synchronous ip.connect()
15 n = 0 : senddata$ = "" : recvdata$ = "" : GOSUB 1000 : GOTO 15
1000 REM connection subroutine
1002 PRINT "connecting... ";
1005 ON SOCKET.SYNC.CONNECT(#0, "192.168.1.205:1000", connect(), send(), recv()), GOTO `unknown,  `ok, `no_open, `no_connect,
`send_err, `recv_err
1010 RETURN
1012 `unknown : PRINT " unknown" : RETURN
1015 `ok : PRINT " success" : RETURN
1020 `no_open : PRINT "can't open" : RETURN
1025 `no_connect : PRINT "no connection" : RETURN
1030 `send_err : PRINT " send error" : RETURN
1035 `recv_err : PRINT " recv error" : RETURN
1040 REM connect function
1045 FUNCTION connect()
1050  PRINT "connect ";
1055 ENDFUNCTION
1060 REM send function
1065 FUNCTION send()
1070  send = 0 : LIF n < 10 THEN senddata$ = STR$(RND(32767)) : n = n + 1 : FPRINT #0, n, senddata$ : PRINT ">";n; : send = 1
1075 ENDFUNCTION
1080 REM recv function
1085 FUNCTION recv()
1090  recv = 0 : FINPUT #0, r, recvdata$ : LIF recvdata$ = senddata$ THEN PRINT "<"; : recv = 1
1095 ENDFUNCTION
Ready
run
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
connecting... connect >1<>2<>3<>4 <<< ESC at line 1090 >>>
Ready
```

Blocking Server

This is an example of a simple synchronous or blocking server. It listens for connections of clients then receives data in the agreed upon format and echoes back what it has received.

Line 1005 uses **an ON GOTO** statement to print the results of the **SOCKET.SYNC.LISTEN( )** function.

The **recv( )** user defined function in lines 1060-1075 receives the value for variables **n** and **recvdata$** from the connected client and returns a non-zero value of one.

The **send( )** user defined function in lines 1080-1095 copies **recvdata$** to **senddata$** and outputs the value for variables **n** and **senddata$** and returns a non-zero value of one.

The **recv( ) / send( )** repeats until the client disconnects then the **SOCKET.SYNC.LISTEN( )** function returns.

Notice how the **SOCKET.SYNC.LISTEN( )** periodically times out and the program prints “no connection”.

```
10 REM test socket.sync.listen()
15 n = 0 : recvdata$ = "" : senddata$ = "" : GOSUB 1000 : GOTO 15
1000 REM listen subroutine
1002 PRINT "listening... ";
1005 ON SOCKET.SYNC.LISTEN(#0, ":1000", connect(), recv(), send()), GOTO `unknown, `ok, `no_open, `no_connect, `recv_err,
`send_err
1010 RETURN
1012 `unknown : PRINT "unknown" : RETURN
1015 `ok : PRINT " success" : RETURN
1020 `no_open : PRINT "can't open" : RETURN
1025 `no_connect : PRINT " no connection" : RETURN
1030 `recv_err : PRINT " recv error" : RETURN
1035 `send_err : PRINT " send error" : RETURN
1040 REM connect function
1045 FUNCTION connect()
1050  PRINT "connect ";
1055 ENDFUNCTION
1060 REM recv function
1065 FUNCTION recv()
1070  recv = 1 : FINPUT #0, n, recvdata$ : PRINT "<";n;
1075 ENDFUNCTION
1080 REM send function
1085 FUNCTION send()
1090  send = 1 : senddata$ = recvdata$ : FPRINT #0, n, senddata$ : PRINT ">";
1095 ENDFUNCTION
Ready
run
listening...  no connection
listening... connect <1><2><3><4><5><6><7><8><9><10> success
listening... connect <1><2><3><4><5><6><7><8><9><10> success
listening... connect <1><2><3><4><5><6><7><8><9><10> success
listening... connect <1><2><3><4><5><6><7><8><9><10> success
listening... connect <1><2><3><4><5><6><7><8><9><10> success
listening...  no connection
listening...  <<< ESC at line 1025 >>>
Ready
```

Non-blocking Client

This is an example of a simple asynchronous or non-blocking client. It is connecting to a server on another device that essentially echoes back what it has received until the connection is terminated.

Line 20 establishes an event handler subroutine starting at line 1020 for **@SOCKET.EVENT[#0]** events.

Line 25 initializes the send data variables as well as a **done** flag variable and then calls the connection subroutine.

Lines 1000-1015 starts the non-blocking connection and then returns.

Lines 30-35 execute a simple program loop that increments variable **a** while checking for the **done** flag variable to be set. When **done** is set the program prints the current value of the incrementing variable **a** and then starts the connection again.

The **@SOCKET.EVENT[#0]** handler subroutine in lines 1020-1055 print the “result” of the **SOCKET.ASYNC.CONNECT** statement and set the **done** flag variable.

The **RUN** shows the connections each interspersed with the incrementing variable **a** current value showing that program execution continues outside of the socket connection.

```
10 REM test socket.async.connect
20 ONEVENT @SOCKET.EVENT[#0],GOSUB 1020
25 n = 0 : senddata$ = "" : recvdata$ = "" : done = 0 : GOSUB 1000
30 a = a + 1 : LIF done = 1 THEN PRINT "a = ";a : GOTO 25
35 GOTO 30
1000 REM connection subroutine
1005 PRINT "connecting... ";
1010 SOCKET.ASYNC.CONNECT #0, "192.168.1.205:1000", connect(), send(), recv()
1015 RETURN
1020 ON @SOCKET.EVENT[#0], GOTO `none,`ok,`no_open,`no_connect,`send_err,`recv_err
1025 RETURN
1030 `none : PRINT "unknown" : done = 1 : RETURN
1035 `ok : PRINT " success" : done = 1 : RETURN
1040 `no_open : PRINT "can't open" : done = 1 : RETURN
1045 `no_connect : PRINT "no connection" : done = 1 : RETURN
1050 `send_err : PRINT " send error" : done = 1 : RETURN
1055 `recv_err : PRINT " recv error" : done = 1 : RETURN
1060 REM connect function
1065 FUNCTION connect()
1070  PRINT "connect ";
1075 ENDFUNCTION
1080 REM send function
1085 FUNCTION send()
1090  send = 0 : LIF n < 10 THEN senddata$ = STR$(RND(32767)) : n = n + 1 : FPRINT #0, n, senddata$ : PRINT ">";n; : send = 1
1095 ENDFUNCTION
1100 REM recv function
1105 FUNCTION recv()
1110  recv = 0 : FINPUT #0, r, recvdata$ : LIF recvdata$ = senddata$ THEN PRINT "<"; : recv = 1
1115 ENDFUNCTION
Ready
run
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
a = 3091
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
a = 6186
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
a = 9280
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
a = 12376
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10< success
a = 15471
connecting... connect >1<>2<>3<>4<>5<>6<>7<>8<>9<>10 <<< ESC at line 30 >>>
Ready
```

Non-blocking Server

This is an example of a simple asynchronous non-blocking server. It listens for connections from client devices and essentially echoes back what it has received until the connection is terminated.

Line 20 establishes an event handler subroutine starting at line 1020 **for @SOCKET.EVENT[#0]** events.

Line 25 initializes the send data variables as well as a **done** flag variable and then calls the connection subroutine.

Lines 1000-1015 starts the non-blocking listening connection and then returns.

Lines 30-35 execute a simple program loop that increments variable **a** while checking for the **done** flag variable to be set. When **done** is set the program prints the current value of the incrementing variable **a** and then starts the connection again.

The **@SOCKET.EVENT[#0]** handler subroutine in lines 1020-1055 print the "result" of the **SOCKET.ASYNC.LISTEN** statement and set the **done** flag variable.

The **RUN** shows the connections each interspersed with the incrementing variable's current value showing that program execution continues outside of the socket connection.

```
10 REM test socket.async.listen
20 ONEVENT @SOCKET.EVENT[#0],GOSUB 1020
25 n = 0 : senddata$ = "" : recvdata$ = "" : done = 0 : GOSUB 1000
30 a = a + 1 : LIF done = 1 THEN PRINT "a = ";a : GOTO 25
35 GOTO 30
1000 REM connection subroutine
1005 PRINT "listening... ";
1010 SOCKET.ASYNC.LISTEN #0, ":1000", connect(), recv(), send()
1015 RETURN
1020 ON @SOCKET.EVENT[#0], GOTO `none,`ok,`no_open,`no_connect,`send_err,`recv_err
1025 RETURN
1030 `none : PRINT "unknown" : done = 1 : RETURN
1035 `ok : PRINT " disconnect" : done = 1 : RETURN
1040 `no_open : PRINT "can't open" : done = 1 : RETURN
1045 `no_connect : PRINT "no connection" : done = 1 : RETURN
1050 `send_err : PRINT " send error" : done = 1 : RETURN
1055 `recv_err : PRINT " recv error" : done = 1 : RETURN
1060 REM connect function
1065 FUNCTION connect()
1070  PRINT "connect ";
1075 ENDFUNCTION
1080 REM send function
1085 FUNCTION send()
1090  send = 1 : senddata$ = recvdata$ : FPRINT #0, n, senddata$ : PRINT ">";
1095 ENDFUNCTION
1100 REM recv function
1105 FUNCTION recv()
1110  recv = 1 : FINPUT #0, n, recvdata$ : PRINT "<";n;
1115 ENDFUNCTION
Ready
run
listening... no connection
a = 33407
listening... connect <1><2><3><4><5><6><7><8><9><10> disconnect
a = 41115
listening... connect <1><2><3><4><5><6><7><8><9><10> disconnect
a = 44172
listening... connect <1><2><3><4><5><6><7><8><9><10> disconnect
a = 47228
listening... connect <1><2><3><4><5><6><7><8><9><10> disconnect
a = 50284
listening... connect <1><2><3><4><5><6><7><8><9><10> disconnect
a = 53340
listening... connect  disconnect
a = 53832
listening...  <<< ESC at line 30 >>>
Ready
```

ACS Basic Examples

Here are a few sample programs that illustrate the various ACS Basic language features and what can be done with a few lines of code.

Setting the Real-Time Clock

Set the Color LCD's Real-Time-Clock with this short program. The program prompts for the values of the Month, Date, Year, Hour, Minute and Second while range checking the values, then displays the formatted time on the connected ANSI terminal once a second.

```
10 REM set the color lcd real-time clock
15 INPUT "set the RTC first (y/n):", s$
20 IF s$="y" THEN 35
25 IF s$="Y" THEN 35
30 GOTO 155
35 INPUT "month (1-12):", m
40 IF m <1 THEN 35
45 IF m >12 THEN 35
50 @MONTH=m
55 INPUT "date (1-31):", d
60 IF d <1 THEN 55
65 IF d >31 THEN 55
70 @DATE=d
75 INPUT "year (0000-9999):", y
80 IF y <0 THEN 75
85 IF y >9999 THEN 75
90 @YEAR=y
95 INPUT "hour (00-23):", h
100 IF h <0 THEN 95
105 IF h >23 THEN 95
110 @HOUR=h
115 INPUT "minute (00-59):", m
120 IF m <0 THEN 115
125 IF m >59 THEN 115
130 @MINUTE=m
135 INPUT "second (00-59):", s
140 IF s <0 THEN 135
145 IF s >59 THEN 135
150 @SECOND=s
155 ONEVENT @SECOND,GOSUB 170
160 a=0
165 GOTO 160
170 PRINT CHR$(13),
175 ON @DOW,GOSUB 265,270,275,280,285,290,295
180 ON @MONTH,GOSUB 200,205,210,215,220,225,230,235,240,245,250,255,260
185 PRINT d$+" "+m$+FMT$(" %2d",@DATE)+FMT$(", %02d",@YEAR),
190 PRINT FMT$(" %2d", @HOUR)+":"+FMT$("%02d",@MINUTE)+":"+FMT$("%02d",@SECOND),
195 RETURN
200 m$="???":RETURN
205 m$="JAN":RETURN
210 m$="FEB":RETURN
215 m$="MAR":RETURN
220 m$="APR":RETURN
225 m$="MAY":RETURN
230 m$="JUN":RETURN
235 m$="JUL":RETURN
240 m$="AUG":RETURN
245 m$="SEP":RETURN
250 m$="OCT":RETURN
255 m$="NOV":RETURN
260 m$="DEC":RETURN
265 d$="SUN":RETURN
270 d$="MON":RETURN
275 d$="TUE":RETURN
280 d$="WED":RETURN
285 d$="THU":RETURN
290 d$="FRI":RETURN
295 d$="SAT":RETURN
```

Displaying an Animated Radial Gauge

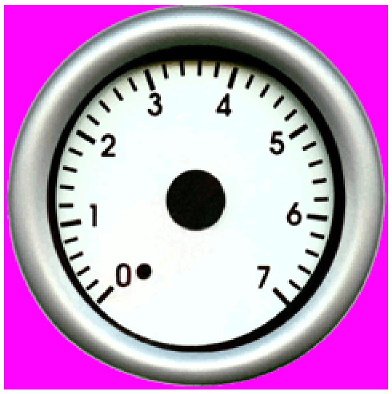
This short program displays a radial gauge whose indicator needle ramps from min to max and then back down. Here's how it works.

In order to make the program somewhat easier to read, a file defining several **CONST**ant variables is **INCLUDE**d on line 15. Here's the contents of included file constants.bas:

```
type constants.bas
CONST False=0,True=1
CONST ScreenWidth=320,ScreenHeight=240
REM Screen Object Types
CONST None=0,Icon=1,Button=2,ToggleButton=3,BackButton=4,Slider=5
CONST Label=6,TouchKeypad=7,RadialGauge=8,LinearGauge=9,Listbox=10
CONST SpinnerKnob=11,Textbox=12
REM Alignment and Transparencies
CONST Left=0,Center=1,Right=2,Top=0,Middle=1,Bottom=2,Solid=0,Transparent=1
REM Drawing Surfaces
CONST Display=0,Work=1,Background=2
REM Screen Events
CONST DestructedEvents=1,ConstructedEvents=2
REM Screen Object Event Mask bits
CONST TouchEvents=1,ValueEvents=2,ReleasedEvents=4
REM Screen Object Events
CONST Touched=1,ValueChanged=2,Released=3
REM Icon Options
CONST NumberOfImages=0,Attributes=1,DisableAdvance=1
REM SpinnerKnob  and Gauge Options
CONST IndicatorLength=0,IndicatorZero=1,IndicatorWidth=2,MinValue=5,MinAngle=6,MaxValue=7,MaxAngle=8
REM + Slider Options
CONST MinOffset=6,MaxOffset=8
REM + Gauge Options
CONST HAlign=3,VAlign=4
REM + Listbox Options
CONST NumberOfItems=0,TopItem=1,ItemHeight=2
REM + Textbox Options
CONST Style=0,NumberOfChars=1
REM + Textbox Styles
CONST NumbersOnly=1,Password=2,Lowercase=4,Uppercase=8,Readonly=16
```

Now that these constant variables are defined in the program, they can be used instead of their numerical equivalents.

Program lines 30-40 configure screen zero, screen object 0 to be a radial gauge type with the defined options. The screen object's **.IMAGE$** is a previously loaded .BMP resource:



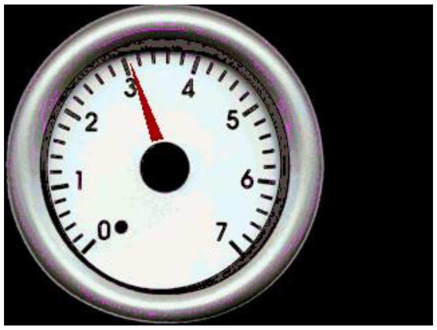
The magenta background color will become transparent when this indexed bitmap is rendered.

Line 50 then starts the screen display system by changing to screen 0. Line 60 waits for the screen to be constructed.

Then lines 70 and 80 set the gauge's value from 0 to 100 and back.

```
10  REM Test Guage Screen Object
15  INCLUDE "Constants.bas"
20  @ANSI.ENABLE=0:@BACKLIGHT=1
30  @SCREEN.OBJ.TYPE[0,0] = RadialGauge
31  @SCREEN.OBJ.OPTION[0,0,IndicatorLength] = 80
32  @SCREEN.OBJ.OPTION[0,0,IndicatorZero] = 20
33  @SCREEN.OBJ.OPTION[0,0,IndicatorWidth] = 10
34  @SCREEN.OBJ.OPTION[0,0,HAlign] = Center
35  @SCREEN.OBJ.OPTION[0,0,VAlign] = Middle
36  @SCREEN.OBJ.OPTION[0,0,MinValue] = 0
37  @SCREEN.OBJ.OPTION[0,0,MinAngle] = 225
38  @SCREEN.OBJ.OPTION[0,0,MaxValue] = 100
39  @SCREEN.OBJ.OPTION[0,0,MaxAngle] = -45
40  @SCREEN.OBJ.IMAGE$[0,0]="Gauge-07-White-TransparentMaskIndexed_240.bmp"
50  SCREENS.CHANGETO 0
60  WAIT @SCREEN.EVENT
70  FOR n = 0 TO 100 STEP 1 : @SCREEN.OBJ.VALUE[0,0] = n : NEXT n
80  FOR n = 100 TO 0 STEP -1 : @SCREEN.OBJ.VALUE[0,0] = n:NEXT n
90  GOTO 70
999  GOTO 999
```

When the program is run, the gauge needle moves from the MinAngle position when the **.VALUE** is zero to the MaxAngle position when the **.VALUE** is 100, then reverses.



Displaying an Animated Analog Clock

This short program displays an old-style analog clock whose hands move to show the Real-Time clock's time every second. Here's how it works.

The clock hands are drawn as elongated 3-sided polygons. Line 25 allocates the polygon vertices coordinate arrays.

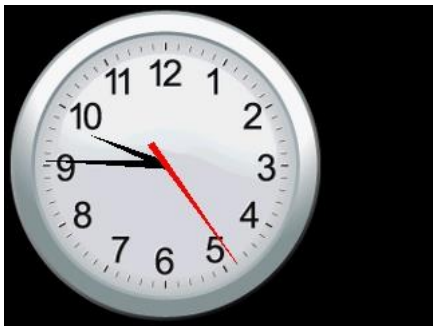
The program minimizes flicker by using double-buffering; static content, the clock face image is drawn on the background surface, then copied to the work surface and dynamic content, the clock hands, are drawn on top. The work and display surfaces are then exchanged for the next update. Line 30 selects the background surface, draws the round clock face using a resource bitmap, then selects the work surface.

Line 35 establishes an event handler that will execute once a second, when the @SECOND system variable changes. Line 40 then executes continuously – except for once a second when the event handler executes.

The subroutine at lines 1000-1040 draws the clock. First the background surface is copied into the work surface. Line 1015 converts the RTC 24 hour time to 12 hour. Line 1020 computes the angle for the hour hand then calls a subroutine to draw it. Line 1025 computes the angle for the minute hand then calls the same subroutine to draw it. Line 1030 computes the angle for the second hand then calls a subroutine to draw it. Line 1035 then exchanges the work and display surfaces and line 1040 returns from the event handler.

The hand drawing subroutines initialize the polygon coordinate arrays for the appropriate hand drawn at a zero angle. The FOR/NEXT loop in lines 2010-2035 rotate the polygon coordinates to the angle for the time value. Line 2040 draws the rotated polygon (clock hand) and line 2045 returns to the event handler.

```
10 REM clock demo
15 @ANSI.ENABLE=0:@BACKLIGHT=1
25 DIM x[3],y[3]
30 @SURFACE=2:DRAW.BITMAP.TRANSP 0,RGB(0,255,0),0,0,"Clockface.bmp":@SURFACE=1
35 ONEVENT @SECOND,GOSUB 1000
40 GOTO 40
1000 REM draw clock
1005 h=@HOUR:m=@MINUTE:s=@SECOND
1010 DRAW.COPY 2,0,0,0,0,239,239
1015 IF h >= 12 THEN h=h-12
1020 a=(450-(h*30)-m/2)%360:r=60:c=RGB(0,0,0):GOSUB 1900
1025 a=(450-(m*6)-s/10)%360:r=90:c=RGB(0,0,0):GOSUB 1900
1030 a=(450-(s*6))%360:r=90:c=RGB(255,0,0):GOSUB 2000
1035 DRAW.TOGGLE
1040 RETURN
1900 REM draw minute or hour hands
1905 x[0]=0:y[0]=-4:x[1]=0:y[1]=4:x[2]=r:y[2]=0
1910 GOTO 2010
2000 REM draw second hand
2005 x[0]=-20:y[0]=-3:x[1]=-20:y[1]=3:x[2]=r:y[2]=0
2010 FOR n=0 TO 2
2015 X=x[n]:Y=y[n]
2020 x[n]=MULDIV(X,COS(a),1024)-MULDIV(Y,SIN(a),1024) + 120
2025 y[n]=MULDIV(X,SIN(a),1024)+MULDIV(Y,COS(a),1024) + 120
2035 NEXT n
2040 DRAW.POLYGON.FILLED x,y,c
2045 RETURN
```



Fixed Length Record File I/O

Here's a short demonstration of the FOPEN, FREAD and FWRITE commands:

```
5 DEL "test.dat"
10 FOPEN #1,20,"test.dat"
15 INPUT "how many records:",n
20 FOR r=0 TO n-1
30 FWRITE #1,r,r,"str"+STR$(r)
40 NEXT r
50 PRINT "reading records..."
60 r=0
70 FREAD #1,r,b,b$
75 IF @FEOF(#1) THEN 1000
80 PRINT "rec:",r,"=",b,",",b$
90 r=r+1:GOTO 70
1000 CLOSE #1
Ready
run
how many records:10
reading records...
rec: 0= 0,str0
rec: 1= 1,str1
rec: 2= 2,str2
rec: 3= 3,str3
rec: 4= 4,str4
rec: 5= 5,str5
rec: 6= 6,str6
rec: 7= 7,str7
rec: 8= 8,str8
rec: 9= 9,str9
Ready
type test.dat
0,"str0"
1,"str1"
2,"str2"
3,"str3"
4,"str4"
5,"str5"
6,"str6"
7,"str7"
8,"str8"
9,"str9"
Ready
```

Error Logging

While developing programs without a serial connection, or for stand alone program monitoring it may be advantageous to record any program errors that occur to the SD card. Then when the program stops running, the SD card can be inserted into a PC card reader and the error that caused the program to stop can be examined. The following code sets up ONERROR to transfer control to line 32000 where an ERRORS.TXT file is opened for appended writing and the causal error message is written at the end of the file:

```
10 REM Error Logging Example
20 ONERROR GOTO 32000
30 A=B/0
32000 OPEN #0,"ERRORS.TXT","a+w"
32005 PRINT #0,ERR$()
32010 CLOSE #0
32015 STOP
Ready
run
STOP in line 32015
Ready
type errors.txt
Divide by zero error in line 30
Ready
run
STOP in line 32015
Ready
type errors.txt
Divide by zero error in line 30
Divide by zero error in line 30
Ready
```

Configuration Editor

This example allows a user with a connected terminal emulator to edit the CFSound-IV configuration settings. Here is how it works:

Line 15 disables the **@MSG$** function so that the **GETCH( )** function will operate as required.

Line 20 initializes the current item and items variables then displays the menu.

Lines 22 through 35 comprise an infinite **WHILE** loop that retrieves information about the current item, shows the item and processes any keys that are pressed.

Lines 8000 through 8615 are the **`` `HandleKeys ``** subroutine that waits for key input using the **GETCH(1)** function then processes the received key to navigate amongst the items, each item's fields, enter and exit editing mode, increment/decrement or default the current item, accept numeric/hexadecimal input for certain item types and allow the program to be exited. In order to accept the arrow cursor keys as input ANSI escape sequences are decoded. The Enter key is used to enter editing mode or exit and save the configuration value.

Lines 10000 through 10160 are the **`` `ShowItem ``** subroutine that displays the currently selected item and field as either being browsed or edited using ANSI escape sequences for inverting the text/background.

Lines 11000 through 11030 are the **`` `DefaultItem ``** subroutine that sets the currently selected item to its default value.

Lines 12000 through 12035 are the **`` `ShowMenu ``** subroutine that does just that.

Lines 13000 through 13070 are the **`` `UpdateField ``** subroutine that updates the currently selected configuration value with the numerically entered value.

Note that the entire program is written without a **GOTO** statement and **`` `label ``** variables are used to call subroutines. Nested block **IF/THEN/ELSE/ENDIF** statements facilitate this style of programming and are indented to show nesting levels – improving the program's readability.

```
10 REM Configuration Editor
15 @MSGENABLE = 0
20 item = 0 : items = @CONFIG.ITEMS : GOSUB `ShowMenu
22 WHILE 1
25  itemType=@CONFIG.TYPE[item]:fields=@CONFIG.FIELDS[item]:min=@CONFIG.MIN[item]:max=@CONFIG.MAX[item]
30  GOSUB `ShowItem : GOSUB `HandleKeys
35 WEND
8000 REM
8005 REM handle ansi terminal key input
8010 REM
8015 `HandleKeys : key = GETCH(1)
8020 REM
8025 REM Ansi escape sequences for arrow keys
8030 REM
8035 IF key = 27 THEN
8040  key = GETCH(1)
8045  IF CHR$(key) = "[" THEN
8050   key = GETCH(1)
8055   REM
8060   REM down arrow key
8065   REM
8070   IF CHR$(key) = "B" THEN
8075    IF editing = 0 THEN item = item + 1
8080    IF item >= items - 1   THEN item = 0
8085   ENDIF
8090   REM
8095   REM up arrow key
8100   REM
8105   IF CHR$(key) = "A" THEN
8110    IF editing = 0 THEN item = item - 1
8115    IF item < 0 THEN item = items - 2
8120   ENDIF
```

```
8125   REM
8130   REM right arrow key
8135   REM
8140   IF (CHR$(key) = "C") AND (editing = 1) THEN
8145    GOSUB `UpdateField
8150    IF fields AND field < fields - 1 THEN field = field + 1
8155   ENDIF
8160   REM
8165   REM left arrow key
8170   REM
8175   IF (CHR$(key) = "D") AND (editing = 1) THEN
8180    GOSUB `UpdateField
8185    IF fields AND field > 0 THEN field = field - 1
8190   ENDIF
8195  ENDIF
8200 ELSE
8205  REM
8210  REM enter key
8215  REM
8220  IF key = 13 THEN
8225   IF editing = 0 THEN
8230    field = 0 : editing = 1
8235   ELSE
8240    GOSUB `UpdateField : @CONFIG.WRITE[item] = 1 : editing = 0
8245   ENDIF
8250  ENDIF
8255  REM
8260  REM plus key
8265  REM
8270  IF (CHR$(key) = "+") AND (editing = 1) THEN
8275   IF fields THEN
8280    IF @CONFIG.VALUE[item,field] < max THEN @CONFIG.VALUE[item,field]=@CONFIG.VALUE[item,field]+1
8285   ELSE
8290    IF @CONFIG.VALUE[item] <= max THEN @CONFIG.VALUE[item] = @CONFIG.VALUE[item] + 1
8295   ENDIF
8300  ENDIF
8305  REM
8310  REM minus key
8315  REM
8320  IF (CHR$(key) = "-") AND (editing = 1) THEN
8325   IF fields THEN
8330    IF @CONFIG.VALUE[item,field] > min THEN @CONFIG.VALUE[item,field]=@CONFIG.VALUE[item,field]-1
8335   ELSE
8340    IF @CONFIG.VALUE[item] > min THEN @CONFIG.VALUE[item] = @CONFIG.VALUE[item] - 1
8345   ENDIF
8350  ENDIF
8355  REM
8360  REM X = exit
8365  REM
8370  IF CHR$(key & 223) = "X" THEN
8375   IF editing = 1 THEN
8380    editing = 0
8385   ELSE
8390    PRINT ""
8395    END
8400   ENDIF
8405  ENDIF
8410  REM
8415  REM R = reset (default)
8420  REM
8425  IF CHR$(key & 223) = "R" THEN
8430   IF editing = 1 THEN
8435    GOSUB `DefaultItem
8440   ELSE
8445    PRINT "  Default entire configuration ? (y/n):";
8450    IF CHR$(GETCH(1) & 223) = "Y" THEN
8455     FOR item = 0 TO items-2 : GOSUB `DefaultItem : NEXT item : item = 0
8460    ENDIF
8465   ENDIF
8470  ENDIF
8475  REM
8480  REM 0 - 9 keys
8485  REM
```

```
8490  IF (key >= 48) AND (key <= 57) AND (editing = 1) THEN
8495   IF (itemType=0) OR (itemType=2) OR (itemType=13) OR (itemType=14) THEN
8500    numberEdit = 1 : IF LEN(number$) < 5 THEN number$ = number$ + CHR$(key)
8505   ENDIF
8510   IF (itemType=12) THEN
8515    hexEdit = 1 : IF LEN(number$) < 2 THEN number$ = number$ + CHR$(key)
8520   ENDIF
8525   IF (itemType=17) THEN
8530    hexEdit = 1 : number$ = CHR$(key)
8535   ENDIF
8540  ENDIF
8545  REM A - F keys
8550  IF ((key & 223) >= 65) AND ((key & 223) <= 70) THEN
8555   IF (itemType=12) THEN
8560    hexEdit = 1 : IF LEN(number$) < 2 THEN number$ = number$ + CHR$(key & 223)
8565   ENDIF
8570   IF (itemType=17) THEN
8575    hexEdit = 1 : number$ = CHR$(key & 223)
8580   ENDIF
8585  ENDIF
8590  REM
8595  REM backspace key
8600  REM
8605  IF (key=8) AND (numberEdit=1) AND (LEN(number$) > 0) THEN number$=LEFT$(number$, LEN(number$)-1)
8610 ENDIF
8615 RETURN
10000 REM
10005 REM Show the current configuration item
10010 REM
10015 `ShowItem : PRINT CHR$(13) + "["; : PRINT item; : PRINT "] - ";
10020 PRINT @CONFIG.NAME$[item] + " = ";
10025 IF fields THEN
10030  IF editing THEN
10035   FOR f = 0 TO fields-1
10040    IF f = field THEN PRINT CHR$(27); "[7m";
10045    IF ((numberEdit = 1) OR (hexEdit = 1)) AND (f = field) THEN
10050     PRINT number$;
10055    ELSE
10060     PRINT @CONFIG.FIELD$[item,f];
10065    ENDIF
10070    IF f = field THEN PRINT CHR$(27); "[0m";
10075    PRINT @CONFIG.SEPARATOR$[item,f];
10080   NEXT f
10085  ELSE
10090   PRINT @CONFIG.VALUE$[item];
10095  ENDIF
10100 ELSE
10105  IF editing THEN
10110   PRINT CHR$(27); "[7m";
10115  ENDIF
10120  IF (numberEdit = 1) OR (hexEdit = 1) THEN
10125   PRINT number$;
10130  ELSE
10135   PRINT @CONFIG.VALUE$[item];
10140  ENDIF
10145  PRINT CHR$(27); "[0m";
10150 ENDIF
10155 PRINT CHR$(27); "[K";
10160 RETURN
11000 `DefaultItem : REM Default a single configuration item
11005 IF fields = 0 THEN
11010  @CONFIG.VALUE[item] = @CONFIG.DEFAULT[item]
11015 ELSE
11020  FOR field=0 TO fields-1:@CONFIG.VALUE[item,field]=@CONFIG.DEFAULT[item,field]:NEXT field:field=0
11025 ENDIF
11030 RETURN
12000 REM
12005 REM Show the menu
12010 REM
12015 `ShowMenu : PRINT "" : PRINT "Configuration Editor" : PRINT ""
12020 PRINT "up/down item |   R = default    | Enter = edit/save"
12025 PRINT "  +/- value  | left/right field |    X = exit"
12030 PRINT ""
```

```
12035 RETURN
13000 REM
13005 REM update field if number entered
13010 REM
13015 `UpdateField
13020 IF numberEdit THEN
13025  IF (LEN(number$)>0) THEN number=VAL(number$) ELSE number=0:IF number < min THEN number=min:IF
number > max THEN number=max
13030  IF fields THEN @CONFIG.VALUE[item,field] = number ELSE @CONFIG.VALUE[item] = number
13035  numberEdit = 0 : number$ = ""
13040 ENDIF
13045 IF hexEdit THEN
13050  IF (LEN(number$) > 0) THEN number=HEX.VAL(number$) ELSE number=0:IF number < min THEN
number=min:IF number > max THEN number=max
13055  IF fields THEN @CONFIG.VALUE[item,field] = number ELSE @CONFIG.VALUE[item] = number
13060  hexEdit = 0 : number$ = ""
13065 ENDIF
13070 RETURN
Ready
```

Basic Revisions

Version	Date	Notes
2.0	Nov-15-2012	First release for new 320x240 display hardware.
2.1	Nov-27-2012	• Added support for zero line number fall through in ON GOTO/GOSUB statements.
2.2	Dec-17-2012	• Added support for optional PS/2 I/O expansion module; added @CONTACT, @CLOSURE and @OPENING system variables and events. • Added support for internal programming application. • Changed ERR$() to return the full error message. • Added setting/clearing of FEOF[#n] after OPEN #n if file is empty.
2.3	Jan-3-2013	• Added @SOUND$ queued sound support. • Added user function capability with FUNCTION/ENDFUNCTION and CALL statements. • Corrected number of files shown in DIR listing. • Corrected FONTS and SCHEMES listings color value display.
2.4	Internal Test Version	• Corrected label screen object text alignment to agree with scheme font alignment. • Added ability to add text label to icon screen object. • Added DRAW.ARC and DRAW.ARC.STYLED commands. • Added DRAW.LINE.DASHED and DRAW.BOX.DASHED commands. • Added SORT command. • Added INCLUDE command. • Changed TEXTWIDTH() and TEXTHEIGHT() to TEXT.WIDTH() and TEXT.HEIGHT(). • Changed HEXVAL() and HEXSTR$() to HEX.VAL() and HEX.STR$(). • Added BITMAP.WIDTH() and BITMAP.HEIGHT(). • Added @CAPTURE and @PWM system variables and events.
2.5	Sep-25-2013	• Added RESOURCES.FLASHERASE command. • Corrected FONTS.LIST and SCHEMES.LIST headers and optional range argument handling. • For label and listbox screen objects added check for object’s scheme’s font if background transparency then don’t draw the background box when no object IMAGE$ resource defined. • Added textbox screen object. • Added support for `labels at the beginning of program lines and as the target of GOTO, GOSUB, etc. • Added support for nested block IF/ELSE/ENDIF. • Added UBOUND() function. • Changed line editor to preserve leading spaces after line numbers to allow code indenting.
2.6	Sep-11-2013	• Internal test revision
2.7	Nov-25-2013	• Changes to allow compilation for either 16 or 32-bit integer values.
2.8	Dec-16-2013	• Internal restructuring of variable representation.
2.9	Mar-12-2014	• Added DHCP support. • Added NUM command for auto-numbering support. • Added command history capability. • Initial DMX512 via Ethernet using Art-Net support. • Initial SMTP send e-mail capability.
3.0	May-7-2014	• Internal restructuring of program source files. • Added @CONFIG.FIELD$[], @CONFIG.SEPARATOR$[] and @CONFIG.ITEMS system variables to facilitate writing configuration editor sample. • Made user functions first class and callable in expressions. • Added SOCKET async and @SOCKET sync functionality. • Added SEARCH, BREAK and CONTINUE commands. • Removed EXITFOR command now superceded by BREAK, added dyadic PRINT USING and FMT$ capability. • Added CHANGE command. • Added FILE.EXISTS() and @FILE.SIZE[] and @FILE.POSITION[]. • Added optional starting position argument to FIND(). • Added configurable USB serial functionality.

3.1	Sep-4-2014	• Corrected handling of directory paths to support nested directories. • Added support for multidimensional arrays up to 3 dimensions. • Changed HEX.STR$() to output upper case hexadecimal digits. • Fixed bugs with FINSERT and FDELETE commands.

Index